**TASK: PA19**
**CDRL: S007**
**28 June 1996**

INFORMAL TECHNICAL DATA

For

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS
(STARS)

# *Domain Architecture-Based Generation for Ada Reuse (DAGAR) Guidebook Version 1.0*

STARS-PA19-S007/001/00
28 June 1996

Data Type: Informal Technical Data

CONTRACT NO. F19628-93-C-0130

Prepared for:
Electronic Systems Center
Air Force Systems Command, USAF
Hanscom, AFB, MA 01731-2816

Prepared by:
Lockheed Martin Tactical Defense Systems
9255 Wellington Road
Manassas, VA 22110

**TASK: PA19**
**CDRL: S007**
**28 June 1996**

Data Reference: STARS-PA19-S007/001/00
INFORMAL TECHNICAL DATA
*Domain Architecture-Based Generation for Ada Reuse (DAGAR) Guidebook*
*Version 1.0*

Distribution Statement "A"
per DoD Directive 5230.24
Authorized for public release; Distribution is unlimited.

This document, developed under the Software Technology for Adaptable, Reliable Systems (STARS) program, is approved for release under Distribution "A" of the Scientific and Technical Information Program Classification Scheme (DoD Directive 5230.24) unless otherwise indicated. Sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA) under contract F19628-93-C-0130, the STARS program is supported by the military services, SEI, and MITRE, with he U.S. Air Force as the executive contracting agent. The information identified herein is subject to change. For further information, contact the authors at the following mailer address:
helpdesk@stars.reston.paramax.com.

**TASK: PA19**
**CDRL: S007**
**28 June 1996**

Data Reference: STARS-PA19-S007/001/00
INFORMAL TECHNICAL DATA
*Domain Architecture-Based Generation for Ada Reuse (DAGAR) Guidebook*
*Version 1.0*

## Principal Author(s):

Carol Klingler 6/28/96

*Carol Klingler* Date

*James Solderitsch, WPL Laboratories, Inc.* Date

## Approvals:

Teri Payton 6/28/96

Program Manager *Teri F. Payton* Date

(Signatures on File)

**TASK: PA19**
**CDRL: S007**
**28 June 1996**

Data Reference: STARS-PA19-S007/001/00
INFORMAL TECHNICAL DATA
*Domain Architecture-Based Generation for Ada Reuse (DAGAR) Guidebook*
*Version 1.0*

## Abstract

This guidebook describes the Domain Architecture-Based Generation for Ada Reuse (DAGAR) domain architecture and implementation development method. DAGAR has been developed to provide a repeatable, documented method for domain architecture engineering. The method is consistent with the Organization Domain Modeling (ODM) domain engineering method and can be used to support the definition of an asset base architecture and implementation of assets within ODM. The EDGE/Ada (Enhanced Domain Generation Environment for Ada) toolset provides integrated support for all phases of DAGAR, including automated system generation based on choices selected by the application engineer.

The primary purposes of this guidebook are to:

- Provide a definitive DAGAR reference document which promotes public understanding of the method and its applicability through in-depth descriptions of DAGAR concepts, processes, and workproducts.
- Provide substantial practical guidance for using the method by describing DAGAR activities and offering examples to get practitioners started.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. AGENCY USE ONLY *(Leave Blank)* | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | 28 June 1996 | Informal Technical Data |

**4. TITLE AND SUBTITLE**
Domain Architecture-Based Generation for Ada Reuse (DAGAR) Guidebook Version 1.0

**5. FUNDING NUMBERS**
F19628-93-C-0130

**6. AUTHOR(S)**
Carol Klingler
James Solderitsch

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Lockheed Martin Tactical Defense Systems
9255 Wellington Road
Manassas, VA 22110-4121

**8. PERFORMING ORGANIZATION REPORT NUMBER**
Document Number
STARS-PA19-S007/001/00

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Department of the Air Force
ESC/ENS
Hanscom AFB, MA 01731-2816

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
S007

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**
Distribution "A"

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

This guidebook describes the Domain Architecture-Based Generation for Ada Reuse (DAGAR) domain architecture and implementation development method. DAGAR has been developed to provide a repeatable, documented method for domain architecture engineering. The method is consistent with the Organization Domain Modeling (ODM) domain engineering method and can be used to support the definition of an asset base architecture and implementation of assets within ODM. The EDGE/Ada (Enhanced Domain Generation Environment for Ada) toolset provides integrated support for all phases of DAGAR, including automated system generation based on choices selected by the application engineer.

The primary purposes of this guidebook are to:

- Provide a definitive DAGAR reference document which promotes public understanding of the method and its applicability through in-depth descriptions of DAGAR concepts, processes, and workproducts.
- Provide substantial practical guidance for using the method by describing DAGAR activities and offering examples to get practitioners started.

**14. SUBJECT TERMS**

**15. NUMBER OF PAGES**
146

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | SAR |

Data Reference: STARS-PA19-S007/001/00
INFORMAL TECHNICAL REPORT
*Domain Architecture-Based Generation for Ada Reuse (DAGAR) Guidebook*
*Version 1.0*

# Table of Contents

Data Reference: STARS-PA19-S007/001/00
INFORMAL TECHNICAL REPORT
*Domain Architecture-Based Generation for Ada Reuse (DAGAR) Guidebook*
*Version 1.0*

# List of Exhibits

Data Reference: STARS-PA19-S007/001/00
INFORMAL TECHNICAL REPORT
*Domain Architecture-Based Generation for Ada Reuse (DAGAR) Guidebook*
*Version 1.0*

# List of Exhibits

# Prologue

This document is version 1.0 of the *Domain Architecture-Based Generation for Ada Reuse (DAGAR) Guidebook*. The guidebook describes the DAGAR domain architecture engineering method. The DAGAR method was developed on the Software Technology for Adaptable, Reliable Systems (STARS) program, funded by the U.S. Department of Defense (DoD) Defense Advanced Research Projects Agency (DARPA).

The principal author of the guidebook were Carol Klingler of Lockheed Martin Tactical Defense Systems and James Solderitsch of WPL Laboratories, Inc. The DAGAR method is based on the method used for developing a domain architecture on the Army/Lockheed Martin STARS Demonstration Project.

We strongly encourage trial use of DAGAR and solicit reader review and comments. To submit comments on the guidebook, to learn more about DAGAR, or to discuss how to obtain support in applying it please contact:

Carol D. Klingler
Lockheed Martin Tactical Defense Systems
9255 Wellington Road
Manassas, VA 22110
Phone: (703) 367-1347
Fax: (703) 367-1389
E-mail: klingler@stars.reston.unisysgsg.com

# Part I: Introduction

## 1.0 Guidebook Overview

This guidebook describes the Domain Architecture-Based Generation for Ada Reuse (DAGAR) domain architecture engineering method. DAGAR has been developed to provide a repeatable, documented method for domain architecture engineering. The method is consistent with the Organization Domain Modeling (ODM) domain engineering method, Version 1.0 [20] and can be used to support the definition of an asset base architecture and implementation of assets within ODM. DAGAR also supports selection of assets for an application. The EDGE/Ada (Enhanced Domain Generation Environment for Ada) toolset provides integrated support for all phases of DAGAR, including automated system generation based on choices selected by the application engineer.

## 1.1 Purpose and Scope

The primary purposes of this guidebook are to:

- Provide a definitive DAGAR reference document which promotes public understanding of the method and its applicability through in-depth descriptions of DAGAR concepts, processes, and workproducts.
- Provide substantial practical guidance for using the method by describing DAGAR activities and offering examples to get practitioners started.

The guidebook is **not** directly intended to do the following:

- Provide detailed guidance on the use of the EDGE/Ada tool to support the DAGAR process.
- Provide extensive justification and rationale for reuse, or domain architecture engineering, in general or for the DAGAR domain architecture engineering approach in particular.
- Define the specific role of DAGAR within an overall software engineering process or life cycle.
- Compare DAGAR with other reuse methods or processes (although it may be useful in helping the reader to draw such comparisons).

## 1.2 Audience

This guidebook is targeted to readers having one or more of the following roles in their organizations:

- *Program/Project Planner* — Responsible for planning the objectives, strategy, processes, infrastructure, and resources for software engineering programs or projects. Interested in incorporating systematic, domain-specific reuse into those programs/projects.
- *Reuse Advocate* — Responsible for keeping abreast of reuse concepts, technology, and trends and promoting the establishment/improvement of reuse capabilities and practices within an organization. Interested in understanding how new concepts, processes, methods, and tools can be applied to accelerate reuse adoption.

- *Process Engineer* — Responsible for defining, instantiating, tailoring, installing, automating, monitoring, administering, and evolving software engineering process models. Interested in defining reuse processes or integrating them with overall life cycle process models.
- *Domain Engineer* — Responsible for scoping and modeling domains of interest to an organization, designing architectures for these domains, and implementing collections of assets ("asset bases") conforming to the architectures that can be reused in multiple application systems. Interested in learning to apply domain engineering concepts, processes, methods, and tools.
- *Application Engineer* — Responsible for developing applications that make use of domain assets. Interested in learning to apply concepts, processes, methods, and tools for selecting assets.
- *Architecture Engineer* — Responsible for developing generic architectures for families of systems ("product lines"). Interested in learning to apply domain architecture engineering to develop a generic architecture that captures the commonality among the systems, while still allowing for variability across the systems.

## 1.3 Benefits

By reading this guidebook, various segments of the audience should be better able to do some or all of the following:

- Understand key DAGAR concepts, including:
  - The DAGAR process flow from Define Asset Base Architecture --> Implement Asset Base --> Apply Asset Base, the reasoning for this flow, and the key process steps involved
  - The importance of an asset base architecture in the development of an asset base for a domain, and the differences between asset base architectures and system architectures.
- Assess and communicate the benefits and implications of applying DAGAR concepts, e.g.:
  - Opportunities where domain architecture engineering may be useful
  - Value of DAGAR methods and processes in relation to other software engineering methods, processes, and tools
  - Value of the EDGE/Ada toolset in supporting the application of DAGAR.
  - Ways DAGAR can be tailored and integrated to meet an organization's needs
- Decide whether or not to explore DAGAR further

In conjunction with further reading, training, and consultation, the guidebook should enable readers to:

- Assess their organization's receptivity to applying DAGAR and the EDGE/Ada toolset
- Initiate, plan, manage, and perform domain architecture engineering project using DAGAR

## 1.4 Guidebook Organization

The body of the guidebook is organized into three major parts:

- **Part I: Introduction**
  - — *Section 1: Document Overview (this section)* — Defines the purpose, scope, and audience of the document.
  - — *Section 2: DAGAR Background* — Describes the factors that motivated the development of DAGAR, the context in which it was developed, and how it has been applied.
  - — *Section 3: DAGAR Core Concepts* — Explains the concepts that provide a foundation for DAGAR.
- **Part II: DAGAR Processes and Products**
  - — *Section 4: DAGAR Life Cycle* — Describes the overall DAGAR process in terms of a process model.
  - — *Section 5: Define Asset Base Architecture*— Describes the DAGAR processes involved in defining the architecture for a domain asset base.
  - — *Section 6: Implement Asset Base* — Describes the DAGAR processes involved in implementing domain assets within the asset base.
  - — *Section 7: Apply Asset Base* — Describes the DAGAR processes involved in selecting assets from the asset base for a particular application by making selections within the asset base architecture.
- **Part III: Applying ODM**
  - — *Section 8: DAGAR as a Supporting Method of ODM* — Describes how DAGAR provides support for the *Define Asset Base Architecture* and *Implement Asset Base* subphases of the ODM domain engineering method.

The guidebook also includes the following supplementary material:

- **Appendix A: DAGAR Process Model** — A graphical view of the DAGAR process model, expressed in the $IDEF_0$ notation. (The $IDEF_0$ diagrams are also embedded within the process descriptions in Part II.)
- **References** — Bibliographic entries for all documents referenced in the guidebook.

## 1.5 How to Read the Guidebook

Each segment of the audience has different needs and interests and will thus benefit most from different portions of the guidebook. All segments of the audience should read Part I to gain a basic understanding of the method. The Program/Project Planner and the Process Engineer should read Part III to gain general insight into how the method can be applied. The Process Engineer and the Domain Engineer should read Part II and the appendices to learn about the DAGAR processes and work products in detail, but from differing perspectives: the Process Engineer to gain insight into tailoring and integrating the DAGAR process to support domain architecture engineering, and the Domain Engineer to learn how the process can be enacted to produce and apply reusable domain assets.

Although the guidebook can be read sequentially from beginning to end, it has been structured to support alternative reading styles. As indicated in the previous paragraph, Part I should be read first, but Parts II and III can be read independently. Furthermore, the sections within Parts II and III need not be read in a strictly sequential order. This is due in part to the fact that the guidebook was developed in accordance with the Lockheed Martin Tactical Defense Systems STARS Process Definition Process, which imposes a clear discipline for structuring and presenting process information. The structure of Part II mirrors the hierarchical structure of the DAGAR process model, as depicted in graphical process trees and $IDEF_0$ diagrams. This structure makes it easy to find and read descriptions of specific portions of the process model. In addition, the sections in Part II have a well-defined internal structure that enables them to be read and understood relatively easily without a global understanding of the process. In practice, however, it is useful to read about a process in conjunction with the processes that surround it, including its parent processes. Further guidance for how to read and interpret the process descriptions is provided in the introduction to Part II.

Another important aspect of the guidebook structure is the treatment of key DAGAR terms and workproducts. When a key term is first introduced in the guidebook, it appears in a ***bold italic*** typeface and is defined at that point.

The DAGAR workproducts and data items (all of which are represented as data flows in the $IDEF_0$ diagrams) appear in a SMALL CAPS typeface wherever they are referenced in the document.

# 2.0 DAGAR Background

## 2.1 Origins

DAGAR is based on an adaptation and implementation of the GenVoca model of hierarchical software decomposition and generation developed by Dr. Don Batory and colleagues at the University of Texas [1], [2], and [5]. The approach has been demonstrated in several domains including data structures, communication protocols and avionics ([3], [13], [4]). The GenVoca implementation in the avionics domain was sponsored by the DARPA Domain Specific Software Architectures (DSSA) program.

The word GenVoca, a partial concatenation of the words Genesis and Avoca (the names of two software generation systems independently developed by Don Batory and Sean O'Malley of the University of Arizona [4]), identifies a software development process which features an expanded view of software systems as hierarchical decompositions. Interpreting software systems as such decompositions is not new. Notable investigators who have presented variations of this strategy include D. Parnas, A. Habemann and J. Goguen ([12], [7] and [6]). But, GenVoca is perhaps the best description of a view and explanation of this approach that appears to be practical and applicable through the use of integrated software generation techniques.

Rather than proposing that software systems be constructed through the use of single generic architectures within which components can be selected at fixed points within the architectural framework (the leaves of a software hierarchy), GenVoca permits a highly tailorable hierarchy where the number of interior branches — the essential structure of the architectural framework — is adjustable along with the components which can be selected to fill slots at the leaves of the hierarchy. GenVoca *components* are defined using parametrization options that are not provided in programming languages in common use today including Ada, C and C++. DAGAR's chosen implementation language is Ada and the EDGE/Ada toolset extends Ada to provide the needed language features. DAGAR components as interpreted by the toolset resemble Ada packages that implement an abstract interface that extends the concept of an Ada package specification.

The other projects that have elected to adopt a GenVoca-like approach, and Batory's own analytical efforts, suggest that the basic properties of GenVoca are domain-independent. Architectural methods which rely on the availability of a large collection of modules to package their coverage of a particular application domain are not easily extended as new features are added to the domain. Such methods are not inherently scalable. Contrastingly, GenVoca (and now DAGAR) offers a generative approach by which a small number of highly parameterizable components can be configured automatically to provide a given set of features for the application domain. The addition of new features means small-scale adjustments to some of the components with resulting application systems generated as required. Thus GenVoca is seen to more easily adjust to shifting requirements and show better scalability.

DAGAR as an interpretation of GenVoca consists of three major elements:

- a formal architecture description where the basic categories (realms) of the architecture are named and related to one another;
- a formal description of the interface for each of the architectural categories (the so-called realm specifications); and
- for each component in each architectural category, a complete definition of how the component implements the basic interface of the category.

## 2.2 Motivation

Software reuse is more effective when systematically planned and managed in the context of specific product lines — families of systems that share functionality. Within a product line, domains (areas of common functionality) are analyzed, and information about the domains is captured, organized, and evolved through encapsulation in domain models and reusable assets. These assets can be reused to develop and evolve systems across the product line. The STARS Organization Domain Modeling (ODM) [20] process is a systematic, yet highly tailorable and configurable methodology for engineering of domain models and reusable assets — *domain engineering*. The ODM modeling life cycle guides the development of descriptive models of legacy systems, artifacts, and experience. These descriptive models are transformed into prescriptive specifications of architecturally-integrated assets. However, ODM does not prescribe a method for developing the domain architecture and implementation, since these methods should be based on organization needs and on the domain chosen. The DAGAR (Domain Architecture-based Generation for Ada Reuse) process was developed by the Loral Defense Systems-East STARS team to meet this need for a domain architecture and implementation method that supports ODM.

## 2.3 Objectives

Specific objectives of DAGAR include:

1) To approach system development from a product-line perspective.

2) To allow both commonality and variability to be included in the domain architecture.

3) To implement assets that are consistent with the domain architecture.

4) To allow the generation of asset instances.

## 2.4 Scope and Applicability

### Where DAGAR Can Be Applied:

- Managers and application engineers are open to using generative techniques for domain asset implementation.
- A domain has been identified where there will be a payoff for creating generic assets and reusing them.
- An Ada (83/95) implementation is desired. Other languages could easily be supported, but are not covered in this guidebook.
- The organization is open to using a layered architectural style. A layered architectural style must be suitable for the domain. This style has been found to be suitable for many domains including avionics, protocol stacks, electronic warfare, and signal processing.

### Where DAGAR May Not Be Applicable:

- DAGAR can not be used unless a suitable domain has been identified and modeled. DAGAR does not include information gathering, domain modeling, etc.
- DAGAR cannot be used if the organization is not committed to performing domain engineer-

ing and to the domain engineering/system engineering distinction.

- DAGAR does not include processes for system engineering, or the integration of assets into the application system.

## 2.5 Relationship to Other Products

Within the STARS program, DAGAR is being developed and refined as part of the Lockheed Martin Tactical Defense Systems STARS *Reuse Whole Product*. The Reuse Whole Product includes a set of reuse support technologies that the Lockheed Martin Tactical Defense Systems STARS team has developed, integrated, or used. As part of the Reuse Whole Product effort, these technologies are being further integrated and augmented with a comprehensive set of training materials and examples to unify the technologies and make them easier for practitioners to use in concert. The Reuse Whole Product concept is inspired by Geoffrey Moore's "Crossing the Chasm" model of technology transition [11].

The Reuse Whole Product component technologies can be viewed as addressing reuse at differing levels, or layers, of abstraction. These layers, from highest to lowest level of abstraction, are termed *Concepts*, *Processes*, *Methods*, and *Tools*. In general, technology choices made at any layer will constrain the choices available at the layers below. The Reuse Whole Product reflects a specific set of technology choices made at each layer, as shown in Exhibit 1.

| Level of Abstraction | Products |
|---|---|
| Concepts | STARS Vision<br>Organon Vision<br>CFRP |
| Processes | CFRP<br>ROSE |
| Methods | ODM &<br>Supporting Methods<br>(including DAGAR) |
| Tools | RLF<br>KAPTUR<br>ReEngineer |

**Exhibit 1.** STARS Reuse Whole Product Technology Layers

The technical concepts framing the Reuse Whole Product stem from the STARS vision of product line-based software development, which integrates the concepts of process-driven, domain-specific reuse-based software engineering, supported by modern tools and environments. This vision is augmented in the Reuse Whole Product by the ***organon*** concept, which is founded on the notion of repositories of codified domain knowledge, coupled with proactive technologies to support the use and evolution of the knowledge.

Within this context, the Reuse Whole Product is scoped specifically to provide integrated process and tool support for the ODM domain engineering life cycle. DAGAR, as a supporting method of ODM, provides a domain architecture engineering and asset implementation method. Major technologies in the Reuse Whole Product besides DAGAR are:

- *STARS Conceptual Framework for Reuse Processes (CFRP)* — The CFRP is a consensus STARS product that provides a conceptual foundation and framework for understanding domain-specific reuse in terms of the processes involved. [22]
- *Reuse-Oriented Software Evolution (ROSE)* process model — A CFRP-based life cycle process model that partitions software development into Domain Engineering, Asset Management, and Application Engineering and emphasizes the role of reuse in software evolution.
- *Organization Domain Modeling (ODM)* methodology — A systematic, yet highly tailorable and configurable methodology for performing domain engineering. ODM provides systematic techniques for identifying and selecting strategic domains of focus within product lines, and incrementally and iteratively scoping the domains to mitigate risk and produce robust, usable domain models. [20]
- *Reuse Library Framework (RLF)* domain modeling toolset — A toolset developed by Unisys and STARS which supports taxonomic domain modeling via semantic network and rule-based formalisms, and features graphical and outline-based model browsers. [21]
- *Capture* domain modeling and legacy management toolset — A toolset developed by CTA and NASA (and now being commercialized by CTA) which graphically supports comparative modeling of system artifacts and domain assets.
- *ReEngineer* reengineering toolset — A toolset developed by Unisys to support the reengineering of legacy systems via fine-grained analysis and abstraction of system structure.

The Reuse Whole Product effort is underway and will continue through fall 1996. Several evolving products and supporting materials will be released and available for trial use during that period (including this guidebook).

Although the above technology choices are sound in the Reuse Whole Product context, DAGAR can be applied in conjunction with a wide variety of other methods and tools that support the domain engineering life cycle. This guidebook has been written to be as independent of other Reuse Whole Product components as possible.

## 2.6 Applications To Date

DAGAR has been applied on the Army/Lockheed Martin Tactical Defense Systems STARS Demonstration Project. STARS has worked with the Army, Navy, and Air Force to sponsor three DoD software engineering projects (termed the STARS "Demonstration Projects") to assess product line development in realistic and familiar contexts. Lockheed Martin Tactical Defense Systems supported the Army STARS Demonstration Project, which was performed by the US Army Communications and Electronics Command (CECOM) Software Engineering Directorate. The project focused on domain engineering and system reengineering.

The demonstration project applied ODM and DAGAR to the production of a domain model, an asset base architecture, and assets for the Emitter Location Processing and Analysis (ELPA) domain. More detailed lessons learned from the demonstration project can be found in [16].

Exhibit 2 provides a pictorial overview of the process followed by the demonstration project and indicates some of the tools used and the workproducts produced during this effort. A tailored version of ODM was used by the demonstration project in completing the three horizontally positioned ovals — domain planning, domain modeling, and domain architecture modeling. The DAGAR process was used to perform the top three vertically arranged ovals — domain architecture specification, domain asset implementation, and system implementation.

Domain planning, as performed by the demonstration project, included defining domain engineering objectives and selecting and scoping the domain.

The second phase of domain engineering on the demonstration project was the development of a domain model. The domain model was based on the logical and physical structure of the domain as reflected in systems which implemented the domain functionality. The demonstration project's domain model included three descriptive representations of the domain — the lexicon model, descriptive stakeholders model, and descriptive features model. These models were produced using the Reuse Library Framework (RLF) tool.

The domain model was transformed into a domain architecture model by making informed trade-offs concerning the features and the physical structure to be supported by the domain architecture. The primary outputs of the domain architecture modeling phase were adapted versions of the three models produced during domain modeling.

As the domain architecture specification phase began, the demonstration project found that the ELPA domain architecture needed to support some degree of variability, in addition to domain commonality. This was particularly important since the ELPA domain architecture was to be inserted in legacy systems where a "one size fits all" approach did not work. The demonstration project chose to develop an ELPA domain architecture specification using a preliminary version of DAGAR which was developed by the project based on GenVoca, since GenVoca supported a variable domain architecture.

As developed by the project, the ELPA domain architecture specification included a high-level architectural specification using the le language along with a set of realm specifications, written in the extended Ada dialect supported by the EDGE/Ada toolset. The ELPA realms were organized

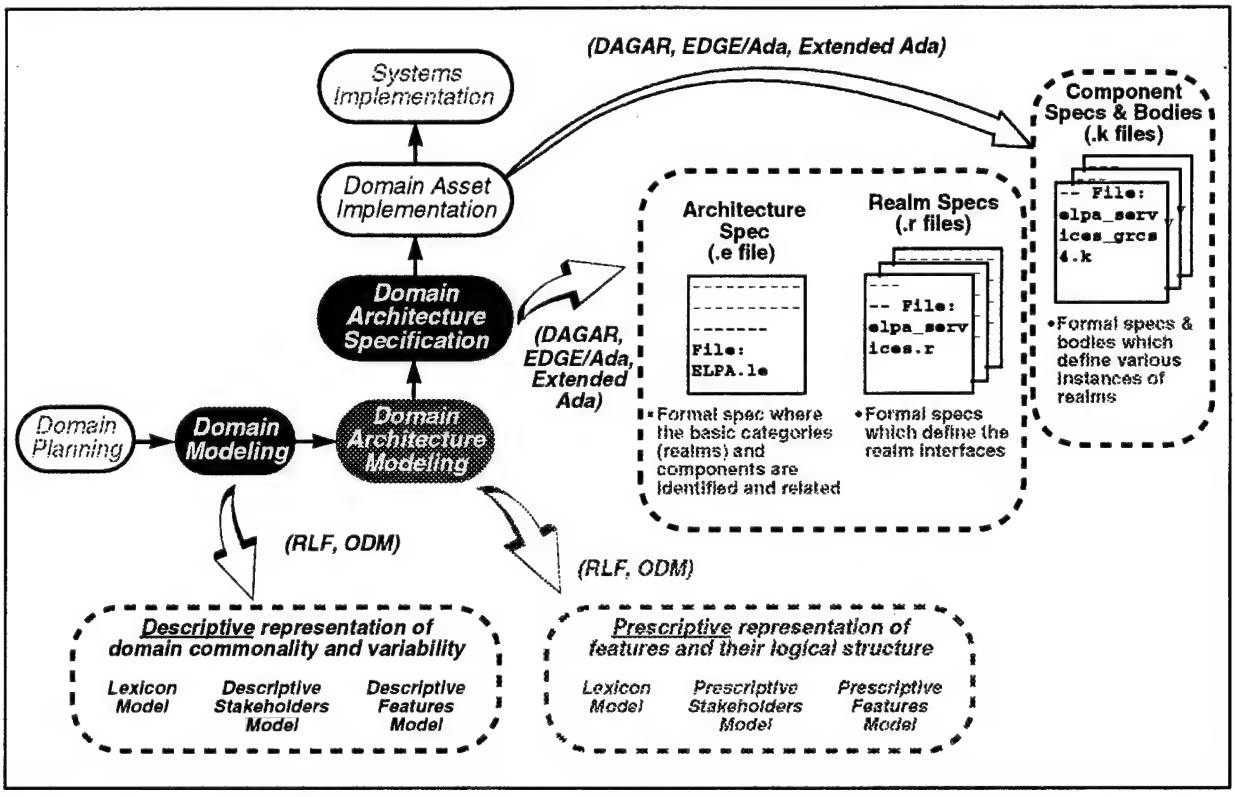



**Exhibit 2.** Demonstration Project Domain Engineering Technical Approach

into several layers including a services layer, a computation layer and a data access layer. The computation layer included realms for fix calculations (a "fix" is fundamental ELPA domain abstraction), coordinate transformations, and required vector and matrix mathematical functions. Early versions of the EDGE/Ada toolset were employed in processing ELPA realms.

Domain asset development on the demonstration project involved the development of component specifications and bodies in accordance with the domain architecture. Component specifications and bodies were also written in the extended Ada dialect supported by the EDGE/Ada toolset. Components were identified based on choices about how to implement each ELPA realm and how services provided in lower level realms could be used to complete the implementation of each ELPA component. Early versions of the EDGE/Ada toolset were employed in processing ELPA components.

The demonstration project is currently performing system integration, applying the asset base to generate domain assets and then integrating these domain assets into a legacy system that uses the domain. This integration will validate the asset base application phase of DAGAR.

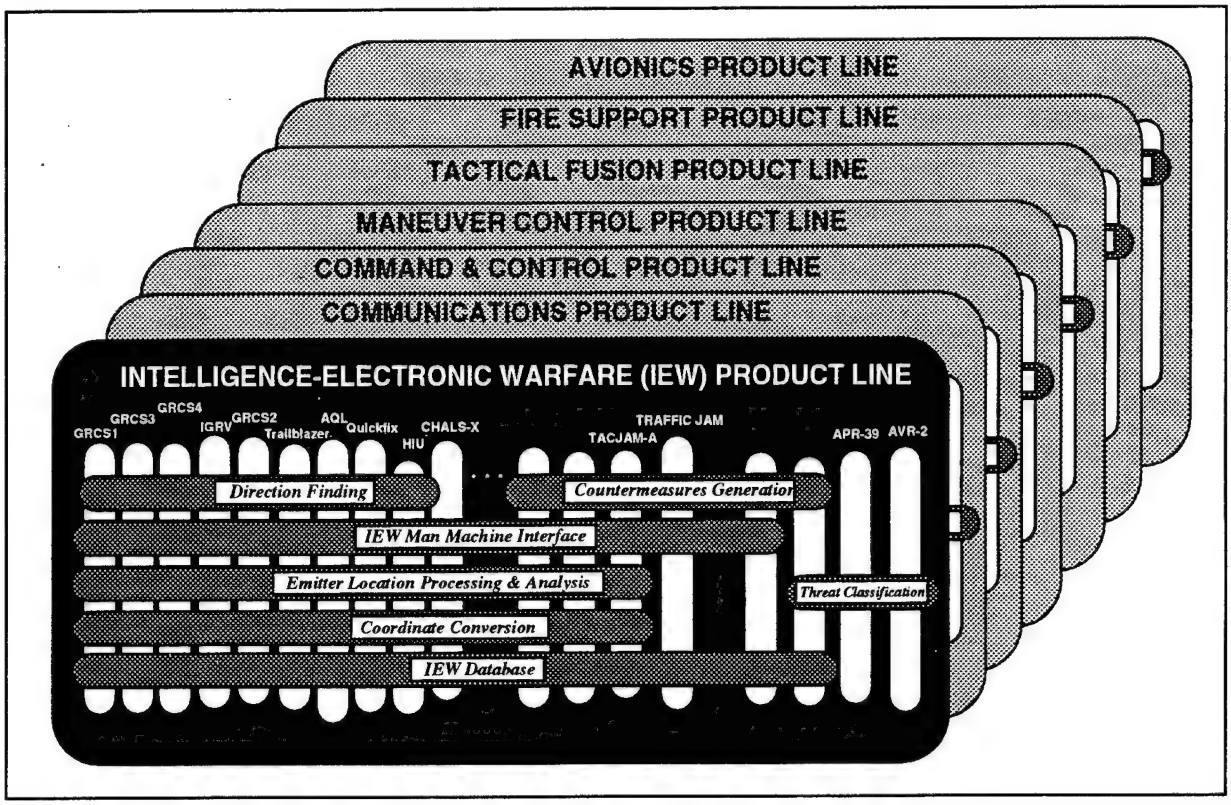



**Exhibit 3.** Army CECOM Software Engineering Directorate Product Lines

# 3.0 DAGAR Core Concepts

To establish a frame of reference for understanding DAGAR core concepts, Section 3.1 defines domain engineering concepts in the context of overall software engineering. Section 3.2 then describes some key definitions needed to understand DAGAR, and Section 3.3 gives an overview of the EDGE/Ada toolset that can be used to support the DAGAR process.

## 3.1 Domain Engineering Concepts

Product-line development involves an approach that emphasizes strategic planning and management of related systems within an application family, based on their inherent commonality and variability. Product-lines provide an explicit boundary and context for the development and reuse of architectures and assets across systems within the product lines. This differs from the more opportunistic approach to reuse, i.e., generic reuse libraries, where the context for the development of reusable assets is either implicit or unknown.

For example, the Army CECOM (Communication-Electronics Command) Software Engineering Directorate is responsible for providing software engineering and evolution support to over 240 Army weapon systems, as shown in Exhibit 3. This support has been organized along seven product-lines: Intelligence-Electronic Warfare (IEW), Communications, Command and Control, Maneuver Control, Tactical Fusion, Fire Support, and Avionics.

Within a product line, domains (areas of common functionality) are analyzed, and information about the domains is captured, organized, and evolved through encapsulation in domain models,

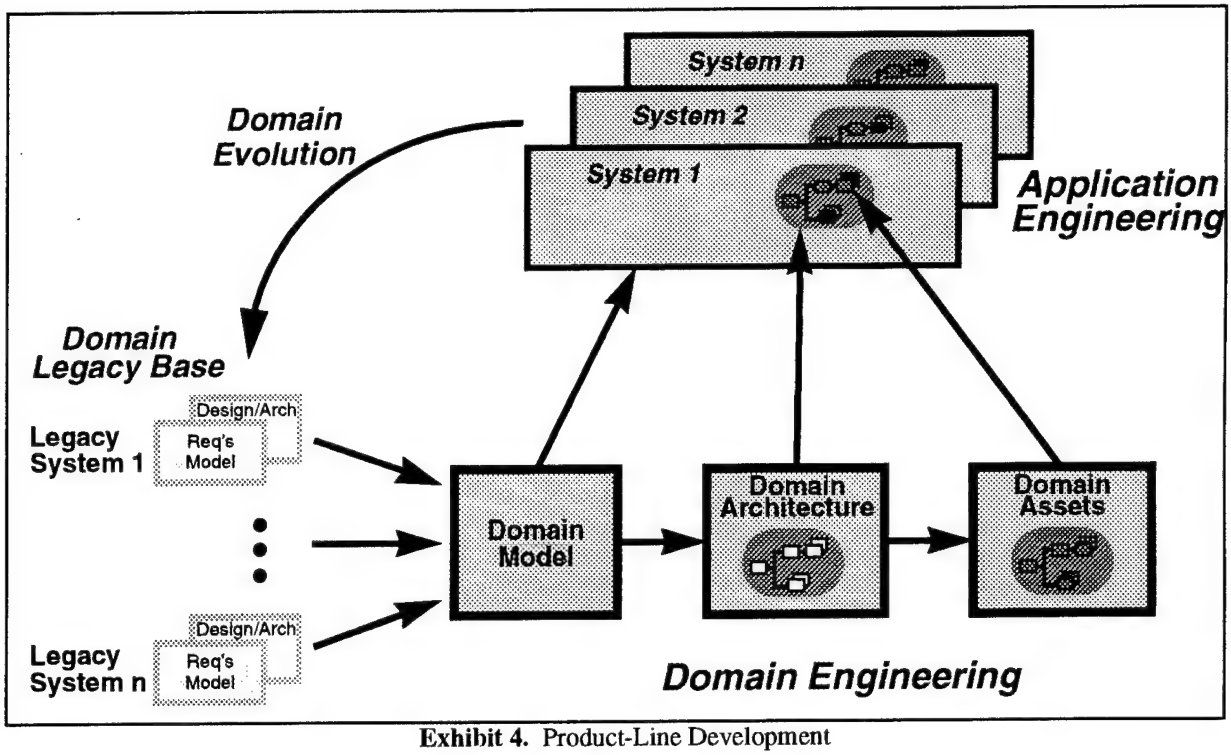



**Exhibit 4.** Product-Line Development

domain architectures, and domain assets. These assets can be reused to develop and evolve systems across the product line. For example, in Exhibit 3 the horizontal bars depict domains in the IEW Product Line, including Direction Finding, IEW Man Machine Interface, and Emitter Location Processing & Analysis. The Army CECOM Software Engineering Directorate is creating domain models, domain architectures, and domain assets for some of these domains.

The vertical bars in Exhibit 3 depict systems that are part of the IEW Product Line, including GRCS1, GRCS3, GRCS4, and IGRV. Where domains intersect systems, the systems include domain functionality. For example, the GRCS1 system (the system at the far left in the picture) includes subsystems comprised of assets from the Direction Finding, IEW Man Machine Interface, Emitter Location Processing & Analysis, Coordinate Conversion, and IEW Database domains, along with system-specific subsystems.

In product-line development, *domain engineering* is performed to engineer and manage each domain. The approach to domain engineering described in this document reflects the ODM process. Exhibit 4 shows how domain engineering is related to the engineering of individual application systems. *Domain engineering* includes developing a domain model, domain architecture, and domain assets. First, a *descriptive domain model* is developed of legacy systems, artifacts, and experience. This descriptive model is transformed into a *prescriptive domain model* that documents the features that the domain architecture will support. A *domain architecture* is then created and represented in a concrete and analyzable format. One important difference between a domain architecture and a system architecture is that a domain architecture must allow for variability, since a domain may include alternate implementations that can be chosen for application systems based on system needs. Finally, *domain assets* are developed that conform to the architecture. Domain assets are not limited to code, but can also include processes, documents, test materials, case studies, presentation materials, etc. Domain assets are managed within an asset base. The asset base can also include tools to aid application engineers in retrieving assets suitable for their application systems.

Once one or more domains have been engineered, *application engineering* of new systems and reengineering of legacy systems can include the use of assets from the domains. Based on system requirements, the application engineer retrieves appropriate assets from the asset base, often assisted by asset retrieval tools. These assets are integrated with the newly developed workproducts (e.g., documentation, code, test cases) for the application system.

## 3.2 Key DAGAR Definitions

This section explains ***realms*** and ***components***, the two basic building blocks of a DAGAR domain architecture. These terms are described based on an Ada implementation of domain assets. For a more general description of realms and components see [1], [2] and [5].

An asset base architecture must balance the need for a common approach to supporting domain requirements with an ability to provide options that admit the necessary domain variability. In building the descriptive and prescriptive feature models (the domain requirements), attention must be paid to representing services and capabilities that should be provided by all products produced from the product line and those that are optionally provided. In addition to such feature variability, the product line should support tuning of products to accommodate various environmental and resource constraints (e.g. memory size, processor power and communication channel band width). The realms and components in a DAGAR architectural approach enables an organization to achieve an effective balance between supporting both commonality and variability. Through the use of realms, DAGAR provides the common ground on which the product line can be produced. Through the use of multiple components for each realm, both feature and performance variability are supported.

A ***realm*** identifies a module[1] and its interface within a layered domain architecture. For example, Exhibit 5 provides a schematic overview of the Emitter Location Processing and Analysis (ELPA) domain architecture as built by the Army STARS Demonstration Project. Each of the named boxes in the figure (e.g., ELPA, Services Layer, ELPA Services, Computational Layer, Fix Calculations, Filtering) is a realm. Each of these realms represents a core set of services provided by assets in the domain.

Each realm contains a ***realm specification***. In an Ada implementation, the realm specification is an Ada package, except that named *holes* are allowed besides Ada declarations. These holes show places in the package specification where implementations may differ in their choices for a declaration, thus allowing variability in the domain architecture. The holes are typically missing Ada type declarations or fragments thereof.

A ***component*** provides one instance of the implementation of a realm. Each realm has one or more components, any one of which can be chosen as the implementation for the realm in an instance of the domain. Each component contains a component specification and a component body.

The ***component specification*** tells how each of the realm holes are to be filled. The ***component body*** provides the Ada package body, the implementation of package functions and procedures. The combination of a realm, with one of its component specifications and the corresponding component body contains a full Ada package specification and body.

---

1. We are using the word *module* here to signify an architectural unit instead of using the traditional word *component*. The word *component* has a special meaning in DAGAR.

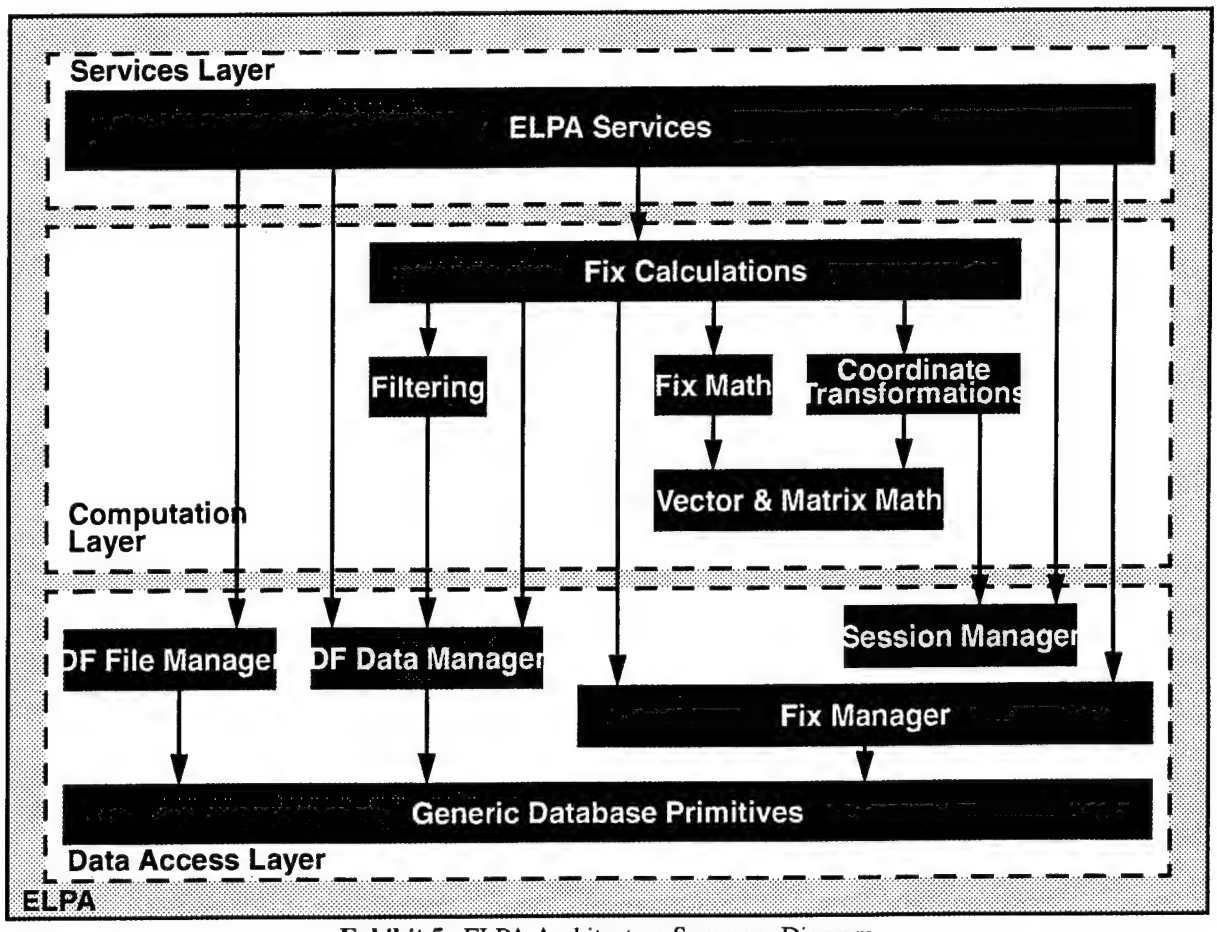



**Exhibit 5.** ELPA Architecture Summary Diagram

Variability occurs in the domain when a realm contains several components, each of which implements the same set of services in a different ways. An application engineer creating an instance of the domain for use in a particular system, can choose the appropriate component for each realm, based on the requirements for the system.

Component specifications and bodies can be dependent on other realms, through calls to functionality in the realm specifications. Since these called realms also have holes and can be instantiated with any one of their components, variability can occur at many levels within the layered domain architecture. Thus many different instance implementations can be generated from the same domain architecture depending on the components chosen for each of the realms within the architecture.

More information about how realms and components are created in the DAGAR process, along with an example of each, will be presented in Sections 5.0 and 6.0.

## 3.3 The EDGE/Ada toolset

The EDGE/Ada (Enhanced Domain Generation Environment for Ada) toolset ([15], [18], [19]) provides full integrated support for the DAGAR process. The toolset can be invoked either through the EDGE/Ada GUI, shown in Exhibit 6, or through the command line. EDGE/Ada can be used by domain engineers to compile architecture specifications, realm specifications, component specifications, and component bodies. The EDGE/Ada GUI includes a utilities pull down

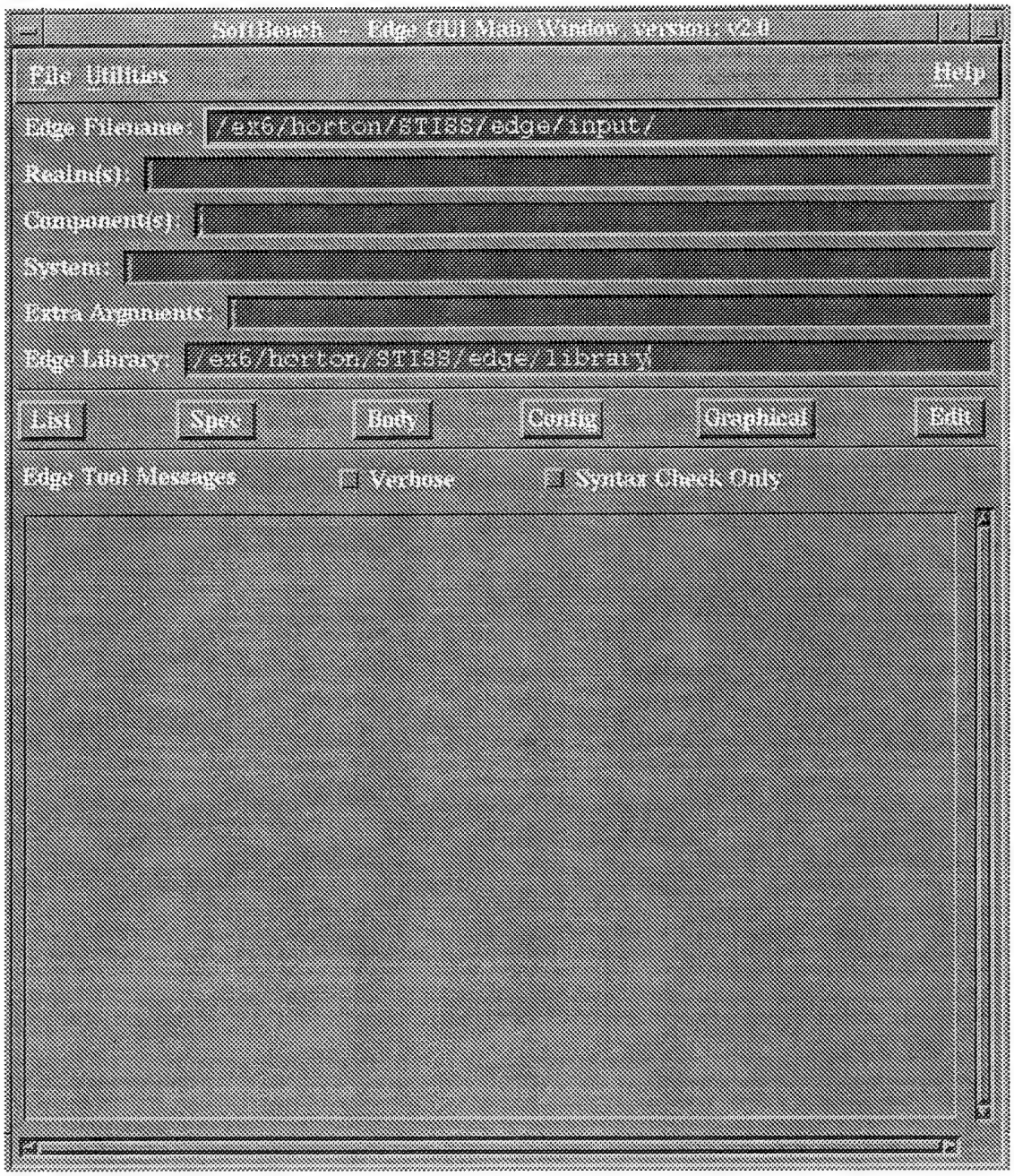



**Exhibit 6.** EDGE/Ada GUI Main Window

menu, shown in Exhibit 7, which provides access to these EDGE/Ada functions used by domain engineers. EDGE/Ada can also be used by application engineers to select components from the domain for use in a particular application system. Once the components have been selected, the domain instance is generated automatically. More information about using the EDGE/Ada toolset in conjunction with the DAGAR process is presented in Sections 4 through 7 below.

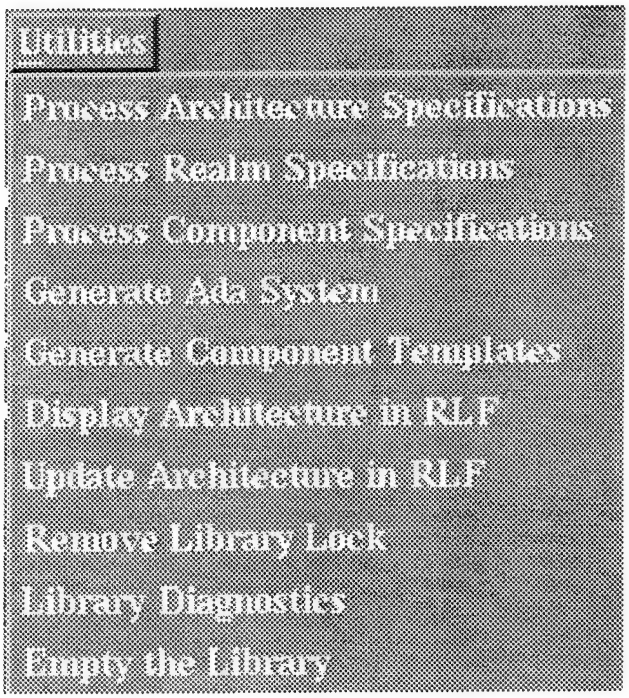



**Exhibit 7.** EDGE/Ada Utilities Pull Down Menu

# Part II: DAGAR Processes and Products

This part of the guidebook describes the DAGAR process model and work products. This introductory portion describes how the part and its sections are organized, defines the conventions used to present information within the sections, and provides guidance in how to read and interpret the information presented.

In general, the process model is described in accordance with the notations and conventions associated with the Lockheed Martin Tactical Defense Systems STARS Process Definition Process [9].

## Part Structure

This part is organized hierarchically, reflecting the hierarchical structure of the DAGAR process model. The full model hierarchy is shown as a process tree in Exhibit 8. Each node in this tree represents a DAGAR process. The nodes below a given process represent the subprocesses that are carried out in performing that process. The following terminology is used in referring to processes at each decomposition level within the process tree:

- ***Process***: The top level, or "root" node, of the process tree (i.e., *Apply DAGAR Process*), representing the overall DAGAR process.
- ***Phase***: One of the three major components of the DAGAR process: *Define Asset Base Architecture*, *Implement Asset Base*, and *Apply Asset Base*.

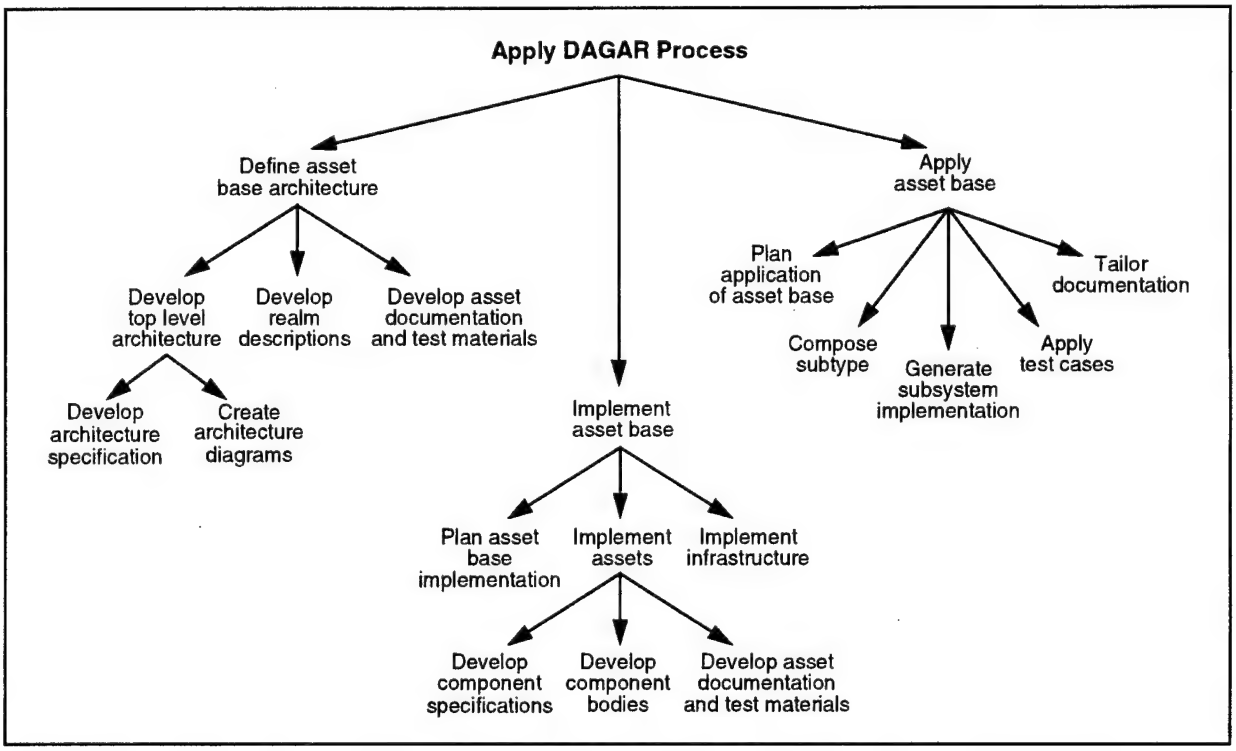



**Exhibit 8.** DAGAR Process Tree

- ***Sub-phase***: A set of tasks that collectively perform a coherent higher-level function within a phase.
- ***Task***: The low-level sets of activities where the detailed DAGAR work is performed and the workproducts are produced.

The overall DAGAR process is described in detail in Section 4. The *Define Asset Base Architecture*, *Implement Asset Base*, and *Apply Asset Base* phases are described in Sections 5, 6, and 7, respectively. The sub-phases and tasks within each of the phases are each described in individual subsections organized hierarchically within Sections 5, 6, and 7.

The phase, sub-phase, and task sections each include a process tree diagram for the current phase, with shaded areas showing which portions of the phase are described by the section. These diagrams not only show the scope of each section, but also serve as recurring road maps to help readers determine their "location" within the process. For example, Exhibit 9 below shows the diagram from Section 5.1, which describes the *Develop Top Level Architecture* sub-phase.

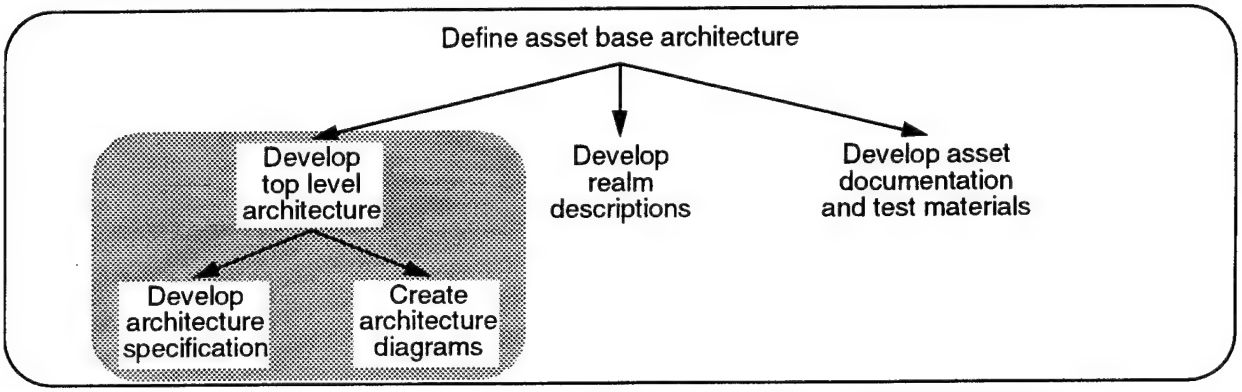



**Exhibit 9.** Example Process Tree Embedded within a Process Description Section

The process trees present a simplified view of the more detailed structure of the process, which is represented in $IDEF_0$ diagrams. The process, phase, and sub-phase sections each include an $IDEF_0$ diagram showing the information flows among the processes at the next lower level. For example, each sub-phase section includes an $IDEF_0$ diagram showing the information flows among the tasks in that sub-phase. Appendix A includes the entire DAGAR $IDEF_0$ process model. If you are unfamiliar with $IDEF_0$, please consult the appendix for a brief introduction to the $IDEF_0$ notation.

The general approach taken within the process description sections is to present information at the lowest section level at which it applies, while minimizing redundancy across sections. For example, low-level details about ODM activities and workproducts are presented in the task sections, whereas information about how the tasks within a given sub-phase interrelate (e.g., sequencing considerations) is presented in the sub-phase section. The higher level sections also discuss the more general and strategic considerations involved in applying DAGAR.

## Section Structure

Section 4, provides an overview of the entire DAGAR process. It places DAGAR within the context of other domain engineering activities. It also introduces the three main phases and the major sequencing options and issues involving these phases.

The phase and sub-phase sections each include the following information:

- The introductory text provides context for the process described in terms of the overall process model and previous and subsequent processes. The overall objectives and benefits of the process are described, as well as (where appropriate) some of the key challenges in carrying out the process.
- **Approach**: The distinctive aspects of the DAGAR approach in meeting the objectives of the process and addressing the challenges described above. Key concepts relevant to the process are introduced here, as are examples to help clarify the concepts.
- **Results**: Describes the resulting workproducts and other outcomes of the process and the potential uses of these results.
- **Process**: An overview of what the process does and, to whatever extent is necessary, how it does it (emphasizing data flows and interactions among the lower level processes, rather than activities that occur within those processes, since they are described in detail in subsequent sections). The description may also introduce concepts needed to understand the lower level processes.
- **Sequencing**: Guidelines regarding the sequence in which lower level processes can be performed. In general, DAGAR processes need not be performed in a particular sequence. Issues addressed here might include:
  - — Different orders in which the processes may be carried out.
  - — Iteration: Lower level processes may be repeated in a "looping" cycle which has some criteria for termination.
  - — Consequences of skipping processes; e.g., starting the sequence in the middle, or terminating the sequence before the end.
  - — Parallelism: This can take various forms: e.g., multiple teams work different lower-level processes in parallel, with possible periodic hand-offs of information; or one group performs the processes in a highly interleaved way.

The Process and Sequencing discussions typically elaborate on the process tree and $IDEF_0$ diagrams included with the section or use them to illustrate points about the process.

The task sections (e.g., 5.1.1, *Develop Architecture Specification*) constitute the bulk of the process model description. These sections describe the low-level DAGAR activities, workproducts, and information flows in detail and also offer tactical guidance for regulating and controlling the processes. The task sections are organized as into the following segments:

- The first paragraphs set the process context for the task (where have we come from, where are we going) and outline the task's key objectives and challenges, and the distinctive aspect of the task within the DAGAR process.
- **Approach**: Describes the DAGAR approach to accomplishing the task objectives. Key concepts are introduced and illustrated where appropriate with examples related to the example domain.
- **Workproducts**: Key results of the task. The value of each workproduct is described within the context of the ongoing DAGAR process.

The three segments above provide a good overview of the task for general comprehension. The remaining segments are targeted more directly to the practitioner performing the task:

- **When To Start**: A set of conditions that should be satisfied before the task can begin.
- **Inputs**: Information that the task accesses and manipulates in performing its function.
- **Controls**: Information that regulates or controls how the task is performed.
- **Activities**: Specific actions or steps to perform in carrying out the task and producing the workproducts. In general, the activities need not be performed strictly in the order they are listed, although this can vary significantly from task to task.
- **When to Stop**: A set of conditions for determining when the task is completed.
- **Guidelines**: Hints, suggestions, and criteria for performing the task effectively. Validation and verification criteria and techniques may also be provided here to determine the completeness, consistency, or quality of the workproducts. (These may also appear in the Activities sub-section when they are best described as distinct activities.)

The Inputs, Controls, and Workproducts directly reflect the inputs, controls, and outputs on the $IDEF_0$ diagrams.

Examples are woven throughout the text, at whatever level they are deemed most useful. They are set off in distinct paragraphs with the key-word Example:

### Presentation Conventions

A number of conventions were applied in writing the sections within this part to make the process model descriptions easier to read and understand. These conventions include:

- **Section subheadings**:

  All of the first-level section subheadings (Approach, etc.) are in bold on a line by themselves. For example:

  **Approach**

  These subheadings are also in a slightly larger font than regular text paragraphs.

- **Information under section subheadings**:

  — Approach, Results, and Process:

  The information under these subheadings is prose paragraphs, in whatever format is most appropriate for the material.

  — Workproducts:

  Each workproduct produced by the process is signified by a subheading on a line by itself in the following format:

  ■ ARCHITECTURE SPECIFICATION

  Substructure within the workproducts is generally shown using bulleted and sub-bulleted items underneath the subheadings.

— Inputs and Controls:

These are presented as bulleted lists. Each item in the list includes the $IDEF_0$ data item name in an underlined run-in heading, followed by some explanation of how the item relates to the process. E.g.:

• ARCHITECTURE SPECIFICATION. This process uses the ARCHITECTURE SPECIFICATION to . . .

— Activities:

Each activity associated with the process is signified by a subheading on a line by itself in the following format:

➤ Identify realms

— Guidelines, When to Start, When to Stop:

These are also presented as bulleted lists. Each item typically includes an underlined phrase (usually a run-in heading, but not always) that concisely summarizes the item, followed by explanatory text. E.g.:

• Iterate back to Asset Base Modeling as necessary. If the domain engineering . . .

Under these subheadings, there may also be non-bulleted prose paragraphs providing sweeping or summary guidelines.

— Examples, Caveats, Notes, etc.:

These are special paragraphs, indented from the surrounding text, which provide example-related material, caveats, notes, or other special information. These usually begin with the word "Example," "Caveat," etc.

- **Typographical conventions**:

— SMALL CAPS (WITH INITIAL LARGE CAPS): DAGAR workproducts (i.e., any workproduct produced directly by a DAGAR process);

— ALL SMALL CAPS: Other $IDEF_0$ data items (net inputs to the overall process);

— *Italic With Initial Caps*: DAGAR process names (phase, sub-phase or task). Activities are generally referred to only within their own task description section, and with no special typographic conventions.

— ***bold italic***: An instance of a key term (usually this is reserved for the initial instance of the term within some cohesive portion of the guidebook, such as a section or group of sections).

— *italic*: Emphasizes or highlights any term or phrase in running text.

Note that **bold regular** highlighting is used only in section headings, subheadings, or run-in headings within paragraphs (with the exception of this sentence).

# 4.0 The DAGAR Process

This section introduces the basic DAGAR process scenario and some of the key terminology used throughout the remaining sections in Part II. The basic sequence described in this document is a single-project scenario, producing an architecture and assets for a single domain of focus, and reusing assets during the development of an application system.

## The DAGAR Context

DAGAR involves the developing an asset base architecture and asset base assets for a domain and making use of the assets in the development of application systems. The process tree depicted in Exhibit 8 (in the introduction to Part II above) shows the scope of DAGAR in terms of the processes involved. In the following descriptions, this overall scope is called the ***DAGAR process***.

The DAGAR process takes place in the context of a *reuse program*, which is described more fully in the STARS Conceptual Framework for Reuse Processes (CFRP) model [22]. The CFRP outlines a taxonomy of reuse processes intended to span all reuse-specific aspects of reuse-based software development. The CFRP model (shown at a high level in Exhibit 10) is partitioned into two major sub-models, known as *idioms*: Reuse Management and Reuse Engineering.

Reuse Management processes form a cyclic pattern of activity addressing the establishment and continual improvement of reuse-oriented activities within an organization by emphasizing learning as an institutional mechanism for change.Reuse Management processes include the Reuse Planning, Reuse Enactment, and Reuse Learning process families.

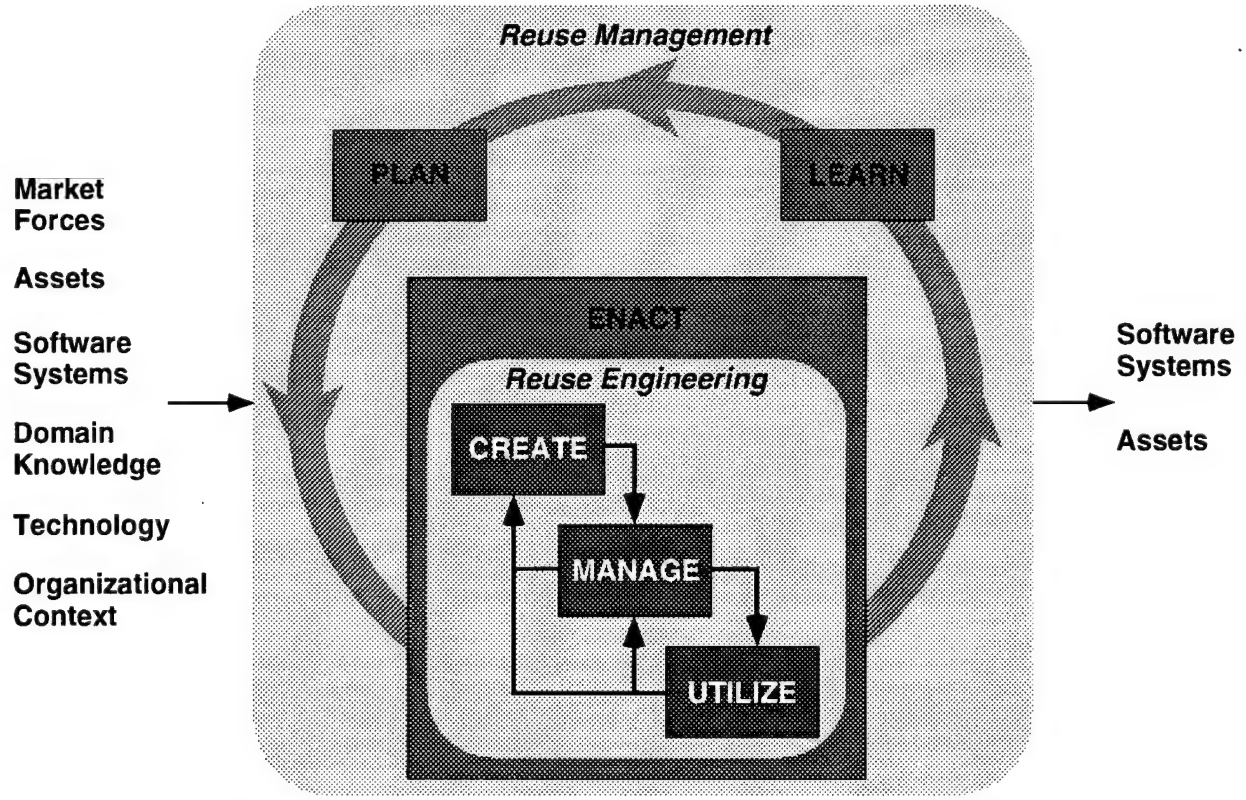



**Exhibit 10.** STARS Conceptual Framework for Reuse Processes (CFRP)

Reuse Engineering processes from a "chained" pattern of activity that addresses reuse-related product development and reuse and explicitly recognizes the role of the broker as a mediator between producers and consumers. Reuse Engineering includes the following three process families:

- *Asset Creation* processes produce and evolve domain models, domain architectures, and domain assets, including requirements and architecture assets, application generators, and software components.
- *Asset Management* processes acquire, describe, evaluate, and organize assets produced by Asset Creation processes, make those assets available to utilizers as a managed collection, and provide services to promote and facilitate reuse of the assets.
- *Asset Utilization* processes reuse the assets made available by the Asset Management processes by identifying, selecting, and tailoring desired assets and integrating them to construct application systems within target domain(s).

In CFRP terms, DAGAR focuses on the Asset Creation (i.e., domain engineering) and Asset Utilization process families of the Reuse Engineering idiom, as shown in Exhibit 11. DAGAR addresses part of the Domain Architecture Development process and the entire Asset Implementation process in Asset Creation. The definition of an architecture within CFRP Domain Architecture Development is addressed in the *Define Asset Base Architecture* phase of DAGAR. A generative approach to Asset Implementation is addressed in the *Implement Asset Base* phase of DAGAR. Parts of the CFRP Asset Identification and Asset Selection processes in Asset Utilization are addressed in the *Apply Asset Base* phase of DAGAR. The *Apply Asset Base* phase addresses the portions of these processes that apply to selecting and validating assets from a single asset base. DAGAR does not address searching across multiple asset bases for assets, or tailoring assets and integrating them with the rest of the application system.

Since DAGAR does not include Domain Analysis and Modeling processes, the DAGAR process assumes that domain analysis and modeling and scoping of the asset base have been carried out and certain workproducts are available before DAGAR is begun. Since DAGAR is a supporting method of the Organization Domain Modeling (ODM) domain engineering method [20], ODM directly supports the creation of the workproducts needed to begin DAGAR (as discussed in Section 8.0), but other domain engineering methods could also be used to produce these workproducts.

The DAGAR process does not include the ongoing management of the domain asset base (i.e., the CFRP Asset Management process). Relationships between the DAGAR process and the management of the asset base must be carefully considered in planning and managing the overall reuse program. Discussing these relationships in detail is beyond the scope of this document.

Exhibit 12 shows the $IDEF_0$ context diagram for the DAGAR process. DAGAR requires that an organization has clearly defined its priorities, analyzed and modeled the domain, and made decisions about what features will be implemented in the asset base. The asset base must have been scoped to derive an overall feature profile for the asset base. Asset base scoping also includes characterizing the market for the asset base, that set of application system contexts in which practitioners will potentially utilize domain assets. The ASSET BASE MODEL contains the subset of the features and potential customer contexts described in the domain model that will be supported by the asset base. The ASSET BASE MODEL should include a map between customers and features showing potential customers for features. The ASSET BASE MODEL controls the DAGAR process, since the architecture and assets implemented must match the features to be implemented.

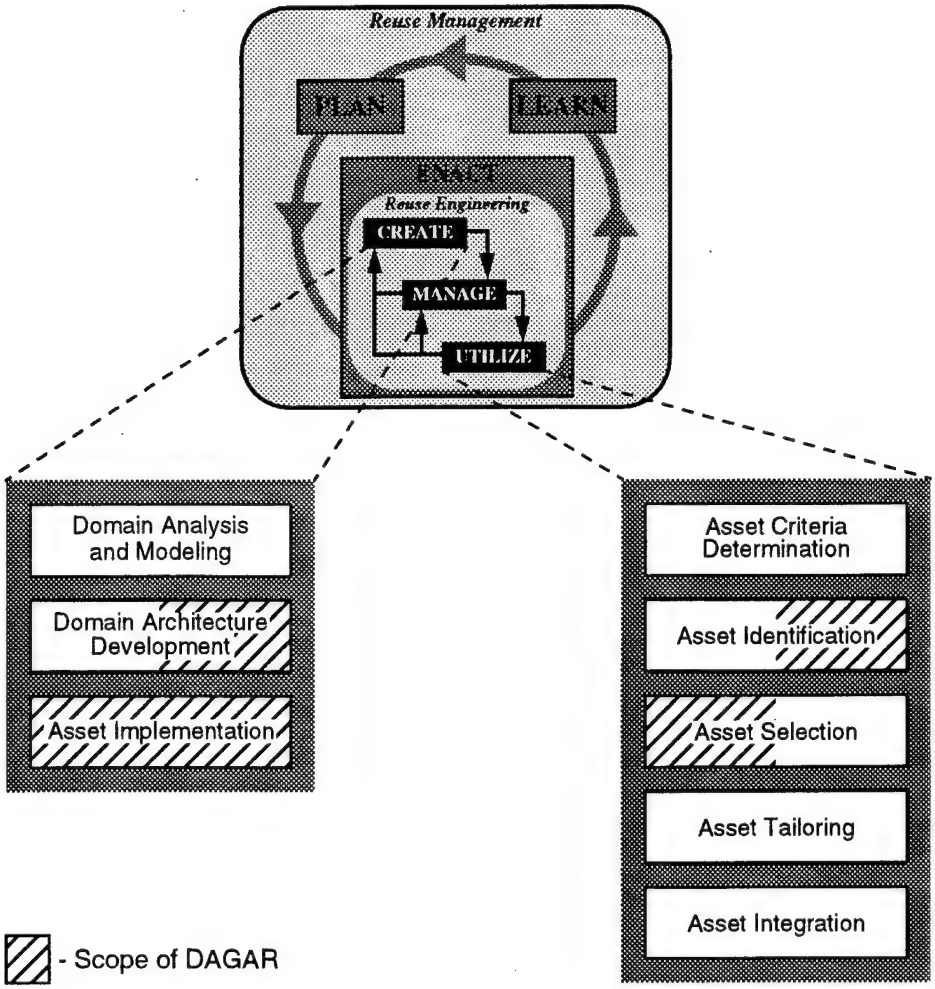



**Exhibit 11.** Scope of DAGAR Processes within the CFRP

An ASSET BASE DOSSIER is also needed to begin the DAGAR process. The ASSET BASE DOSSIER contains information about the customers who will be supported by the asset base. The ASSET BASE DOSSIER also contains usability and feasibility data, as well as traceability to EXEMPLAR SYSTEM ARTIFACTS that can be used in implementing assets. EXEMPLAR SYSTEM ARTIFACTS are artifacts from any stage in the software engineering life cycle for systems or subsystems that are within the scope of the domain functionality. These artifacts can be used as a starting point for developing reusable assets for the domain.

Before an ASSET BASE ARCHITECTURE can be defined, internal and external ARCHITECTURE CONSTRAINTS on the architecture must be identified and understood. Determining external architecture constraints involves addressing external interfaces to asset functionality required by application engineers who will use the asset base. These include constraints imposed by the external system environments in which the assets will operate, as well as external services that may be invoked by asset functions. External architecture constraints can include expected application contexts (e.g. programming language, operating system, subsystem integration concerns, software lifecycle documentation constraints) and hardware platform considerations.

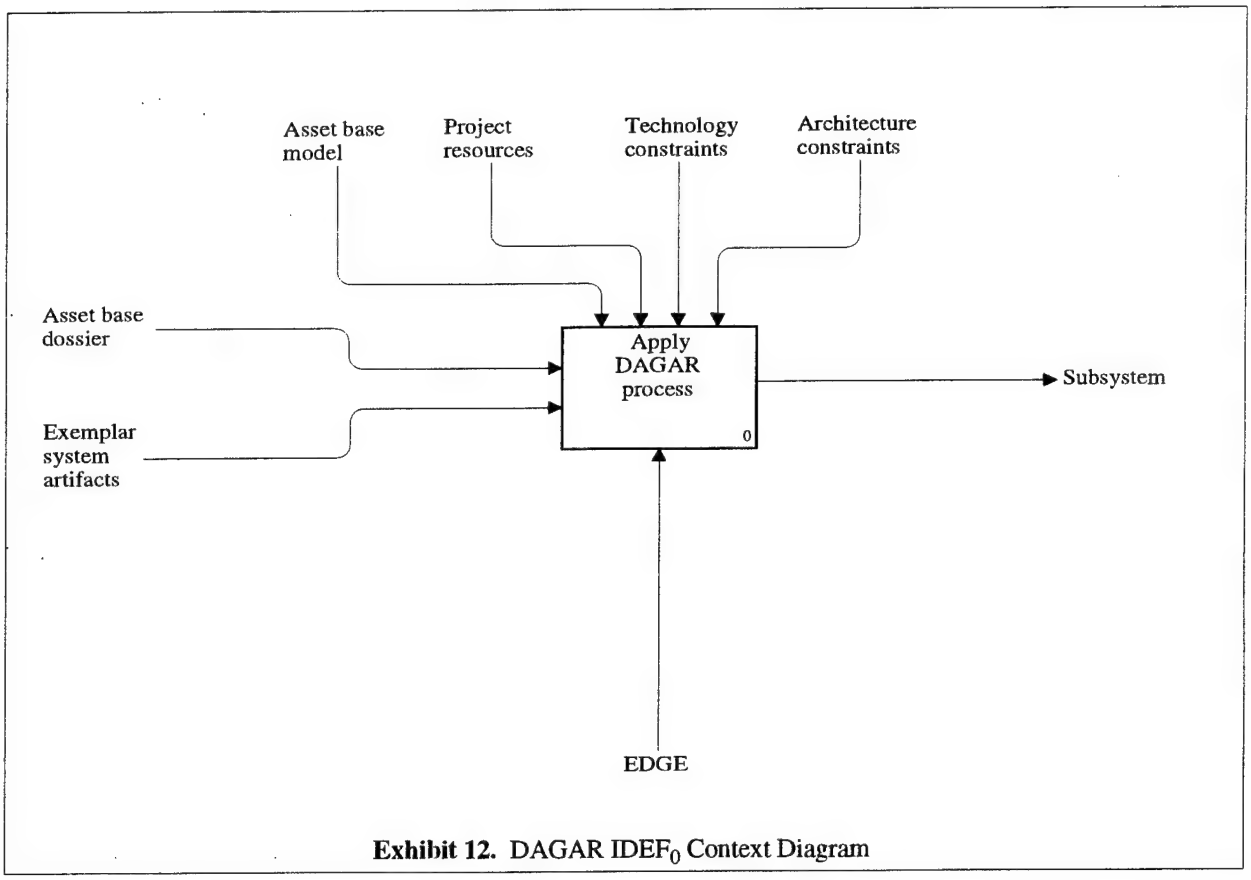



**Exhibit 12.** DAGAR IDEF$_0$ Context Diagram

Determining internal architecture constraints involves addressing issues of how to structure the internal relationships between assets in the asset base. There are two perspectives to be considered here:

- Layering of assets into subsets that can be selected separately by the application engineer. Each of these layerings represents a restricted version of overall domain functionality.
- Asset encapsulations of particular domain functionality – how should the domain functionality be split among assets?

PROJECT RESOURCES and TECHNOLOGY CONSTRAINTS control the *Apply Asset Base* phase of DAGAR. PROJECT RESOURCES are used to develop reasonable objectives and schedules for implementing assets. TECHNOLOGY CONSTRAINTS, such as mandates and limitations, can effect whether assets are used when developing application systems.

SUBSYSTEMS generated from the asset base as the key result of the DAGAR process. These SUBSYSTEMS are then be integrated into application systems.

## Phases of Domain Engineering

The DAGAR process consists of three main ***phases***, as shown in Exhibit 13: *Define Asset Base Architecture*, *Implement Asset Base,* and *Apply Asset Base*. These phases are called (Asset Base) Architecture Definition, (Asset Base) Implementation, and (Asset Base) Application in the following paragraphs for convenience.

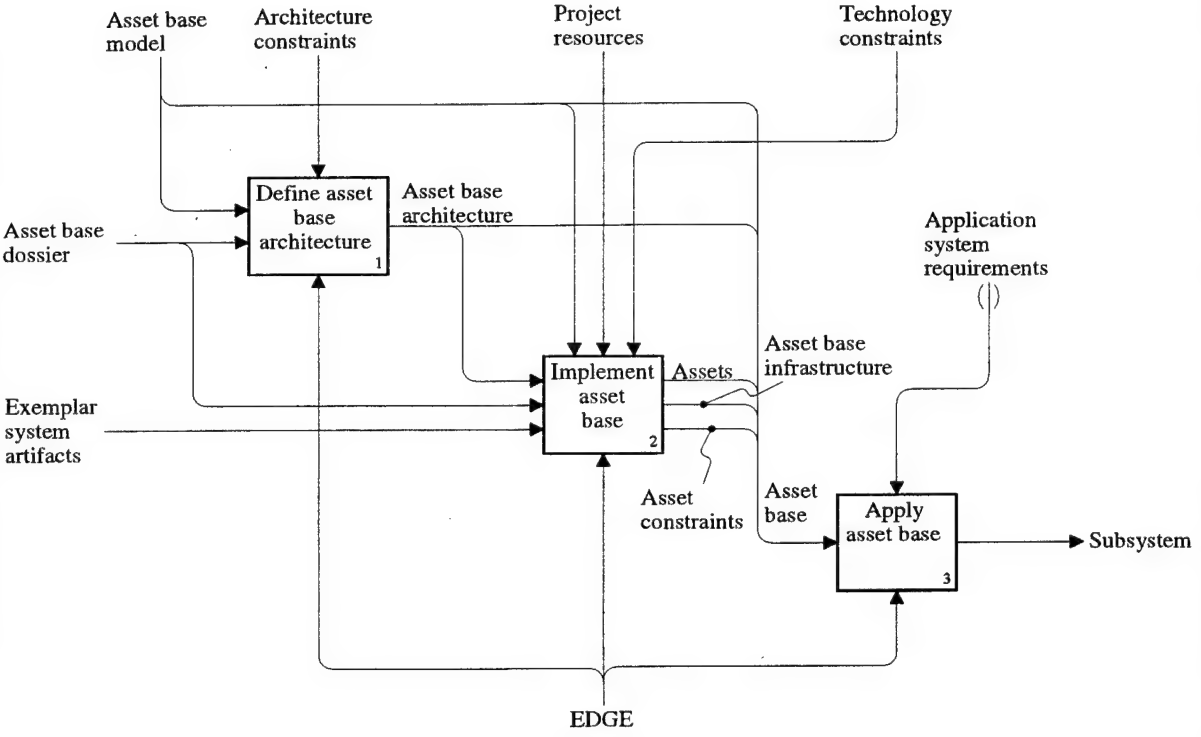



**Exhibit 13.** DAGAR $IDEF_0$ Diagram

The primary purpose of the Architecture Definition phase is to build the ASSET BASE ARCHITECTURE, the architectural framework that forms the foundation for the ASSET BASE. The ASSET BASE ARCHITECTURE includes the range of external interfaces to be supported, definition of the modules within the architecture, and definition of the interconnections among these modules. Architecture documentation and test cases are also developed.

The primary purpose of the Implementation phase of the DAGAR process is to produce an ASSET BASE. An ***asset*** may be a software component, a generative tool, a template for a design document, or any workproduct of the software life cycle specifically engineered for reuse within a domain-specific scope of applicability. The ***asset base*** is the full set of ASSETS for a domain, together with the ASSET BASE ARCHITECTURE that integrates the assets, the ASSET BASE INFRASTRUCTURE required for the domain, and ASSET CONSTRAINTS.

The primary purpose of the Application phase of the DAGAR process is to apply the ASSET BASE to the development of a system that includes functionality compatible with the domain. Application engineers use the ASSET BASE INFRASTRUCTURE and ASSET CONSTRAINTS to choose ASSETS based on system requirements, and generate a SUBSYSTEM for their application. This SUBSYSTEM is then integrated with the rest of the application system.

## Sequencing

Iteration between Architecture Definition and Implementation phases. Once the preliminary ASSET BASE ARCHITECTURE has been developed, ASSET implementation can begin. Expect itera-

tion between these two phases so that the ASSET BASE ARCHITECTURE can be expanded and enhanced based on lessons learned during implementation of the ASSETS.

Implement needed ASSETS first. Implementation should begin with the implementation of ASSETS that are currently needed for application systems being developed. Once these ASSETS have been developed, implementation of other ASSETS can proceed in parallel with using the ASSETS already developed in application systems.

# 5.0 Define Asset Base Architecture

The *Define Asset Base Architecture* phase is the first phase of the DAGAR process. The purpose of the Define Asset Base Architecture phase is for the domain engineer to build the ASSET BASE ARCHITECTURE.

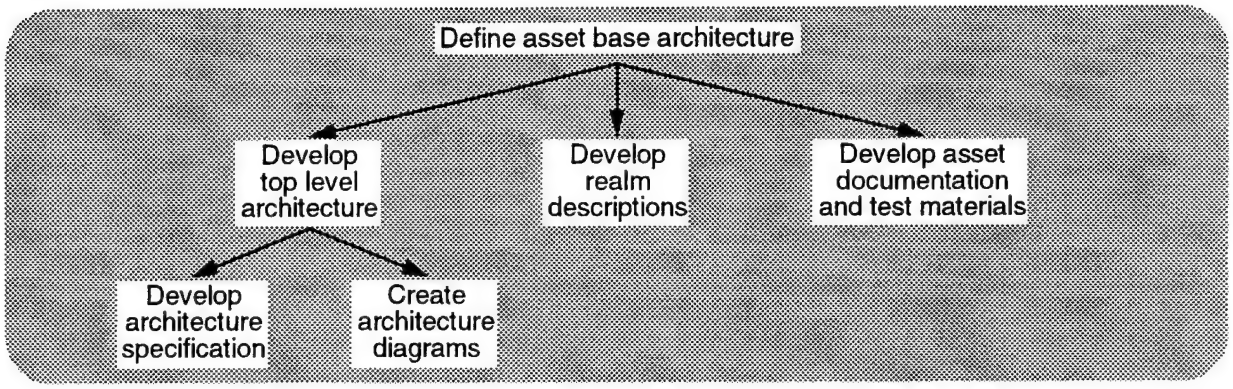



**Exhibit 14.** Define Asset Base Architecture Process Tree

DAGAR requires a thinking about ASSET BASE ARCHITECTURES from a domain perspective rather than a system perspective. Domain engineers need training in DAGAR and asset base architecture development. Domain engineering staff should be provided with the necessary training in DAGAR in order to insure an efficient and knowledgable application of the method. Development of an ASSET BASE ARCHITECTURE requires team members to reevaluate and reinterpret some of their ingrained system design and implementation inclinations. The DAGAR method is based on sound engineering abstractions such as step wise refinement, delayed binding of design decisions and information hiding. However, use of the method requires a mindset adjustment from traditional architecture development to ASSET BASE ARCHITECTURE development that must be learned and adopted by the entire domain engineering team. An asset base architect must be concerned with all possible applications of the ASSETS in a domain. An asset base architect has the responsibility to create a framework expressive enough for the full range of variability in the domain while capitalizing on the commonality.

## Approach

The *Define Asset Base Architecture* phase involves the development of an ASSET BASE ARCHITECTURE — the architectural framework that forms the foundation for the domain assets.The ASSET BASE ARCHITECTURE includes the range of external interfaces to be supported, definition of the modules within the architecture (called REALMS), and definition of the interconnections among these modules. ASSET BASE ARCHITECTURES are distinct from system architectures because ASSET BASE ARCHITECTURES model the range of *variability* in system architectures that can be obtained from the ASSET BASE. Both the EXTERNAL ARCHITECTURE CONSTRAINTS and INTERNAL ARCHITECTURE CONSTRAINTS are taken into account during the *Define Asset Base Architecture* phase.

## Results

The primary output of the *Define Asset Base Architecture* phase is the ASSET BASE ARCHITECTURE, which is used as a framework both for ASSET implementation and for choosing ASSETS during the *Apply Asset Base* phase. In addition to a formal architecture description produced using the EDGE/Ada toolset, this phase also produces less formal supporting material that helps users

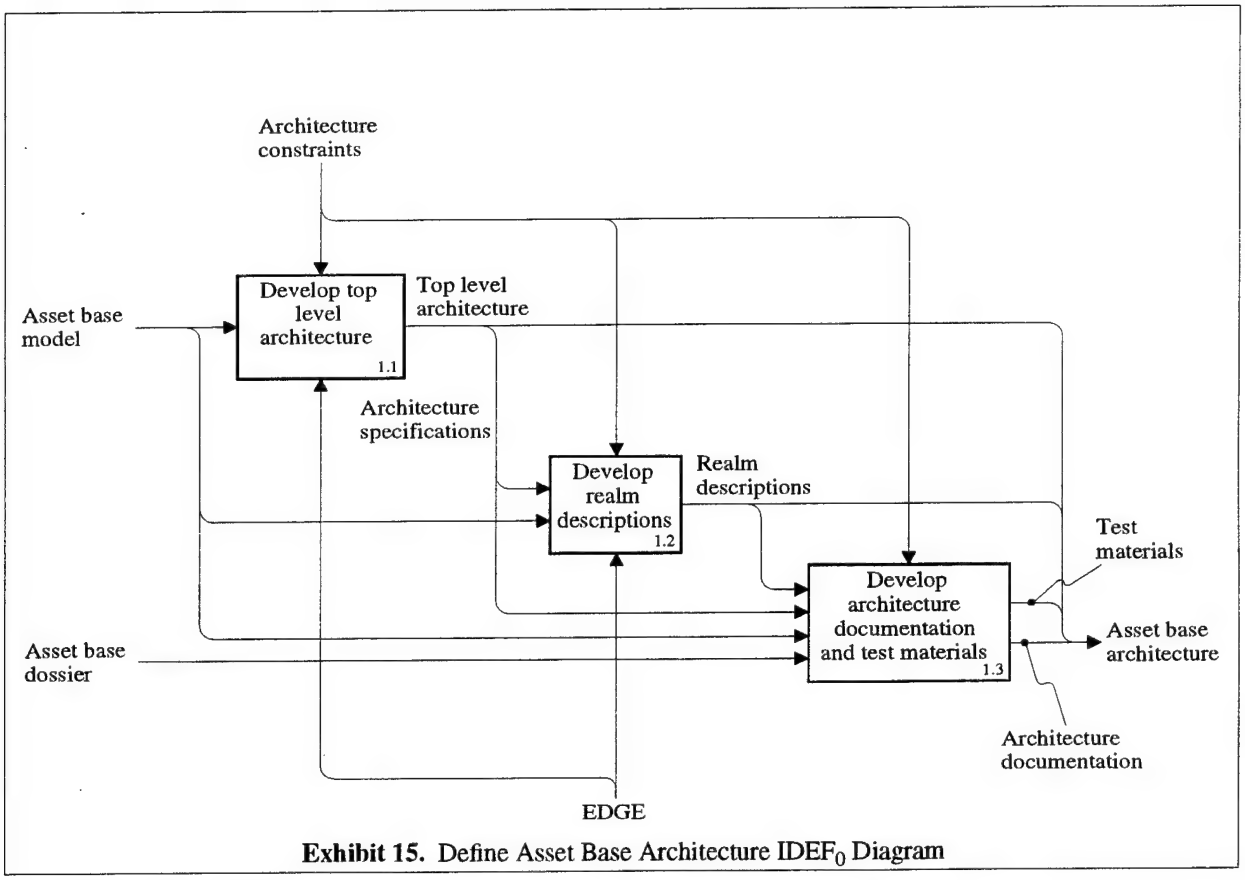



**Exhibit 15.** Define Asset Base Architecture $IDEF_0$ Diagram

understand and use the architecture effectively. The architecture is also used as the framework that is used to select assets when asset base customers interact with the asset base using the Architecture Configuration Assistant (ACA) component of EDGE/Ada.

## Process

There are three main sub-processes in the *Define Asset Base Architecture* phase, as depicted in Exhibit 15:

- In the *Develop Top Level Architecture* sub-phase, domain engineers primarily address the large-scale decomposition and connectivity of the ASSET BASE ARCHITECTURE and use the capabilities of the EDGE/Ada toolset to define and verify the correctness of the evolving architecture specification. In addition, since the architecture specification is defined using a formal architecture specification language (derived from one described in [1]), it is useful to augment the definition with diagrams that help visualize the meaning defined in the specification.

- In the *Develop Realm Descriptions* task, domain engineers go beyond the higher level definition and begin to address the individual services and elements that are contained within each of the major architectural building blocks (***realms***) that make up the architecture. An extended form of Ada is used as the specification language for these building blocks. EDGE/Ada is used to verify that these REALM SPECIFICATIONS for each realm are consistent with the architecture definitions as defined in the ARCHITECTURE SPECIFICATION.

- In the *Develop Architecture Documentation and Test Materials* task, domain engineers set the stage for later usage of the architecture. The architecture is used both to guide the develop-

ment of implementations of each of the realms and to make sure that properties visible within the realm interface description are accurately and faithfully implemented. At the architecture stage, documentation typically takes the form of incomplete templates readable by the preferred documentation tools. Test materials are often at the level of black box test plans and core test data that can be used once components implementing the realms are in place.

## Sequencing

- Iterate back to revise the TOP LEVEL ARCHITECTURE as necessary. Since in practice it is difficult to identify all REALMS and COMPONENTS up front, the domain engineer will likely cycle back to revise the TOP LEVEL ARCHITECTURE as necessary when more aspects of the architecture are identified. The EDGE/Ada tool supports extending the architecture by allowing the processing of ARCHITECTURE SPECIFICATIONS that add new REALMS and COMPONENTS or modify the realm parameters for existing components. It is also likely that the *Develop Top Level Architecture* sub-phase will need to be revisited as the *Implement Asset Base* phase proceeds, to fold in discoveries and address problems that are brought out during implementation and associated activities.

- The subtasks of this phase should not be performed sequentially. All three of the subtasks are best viewed as being alternate points of view into the production of a single architecture. As details emerge within one point of view, these details should then be considered from the other perspectives so that the architecture evolves effectively.

- Toolset-driven development. EDGE/Ada provides significant support for the performance of activities that fall within the *Define Asset Base Architecture* phase. Domain engineers should learn how to use the various capabilities contained in EDGE/Ada and integrate use of the tools into their daily domain engineering activities.

# 5.1 Develop Top Level Architecture

Architecture development should begin with an initial top level consideration of the goals and customers to be served by the production of the asset base. This consideration will naturally start with a first cut at the production of a candidate architecture. This architecture will remain somewhat fluid as architecture definition and asset base implementation proceed. Insights gained during asset implementation will be fed back to architecture development and result in subsequent changes to the architecture.

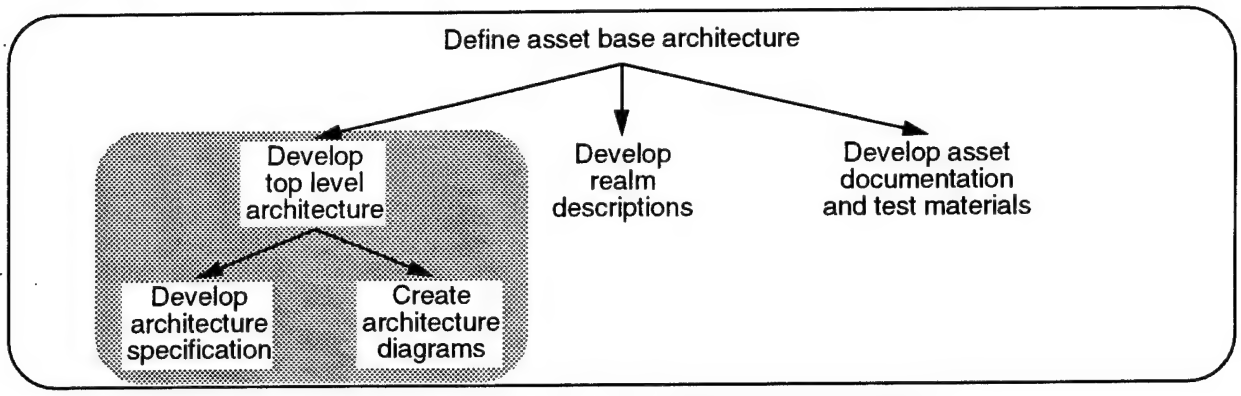



**Exhibit 16.** Develop Top Level Architecture Process Tree

The primary purpose of the *Develop Top Level Architecture* sub-phase of DAGAR is to produce a concrete formulation of the domain architecture, the ARCHITECTURE SPECIFICATION, as supported by the EDGE/Ada toolset. This formal description can be thought of as a high-level layout of the scope and content of the architecture. As described below, the main elements of a DAGAR architecture are the set of realms that encapsulate a set of operational capabilities (sometimes called a virtual machine) and for each realm, a set of components that will be alternate implementations of these capabilities. Each component is free to make use of the services of other realms in defining the details of the implementation of the realm to which it belongs. Each realm used by a component in this way is called a ***realm parameter***. In addition, the top level formal description is supplemented by less formal auxiliary views that are often drawn using simple graphics tools or the drawing modes offered in document preparation software or CASE tools.

It may be difficult for engineers just learning about DAGAR and its EDGE/Ada tool support to approach ASSET BASE ARCHITECTURE design without thinking in systems design terms. An asset base is more than a single system and an asset base architecture is much more than a system architecture. An ASSET BASE ARCHITECTURE represents the potential to design many systems, each with its own distinct system architecture. Each component connects to a fixed number of realms, but each component in a realm can have quite distinct parameter profiles. Each system may require selecting components from some, or all, of the existing realms. The domain engineer must create the architecture to allow for multiple users with multiple systems and requirements to make use of the assets presented through the architecture. This multi-system thinking requires effective use of the information accumulated during domain analysis including an understanding of the expected customers and their needs.

### Approach

The task of architecture definition begins with a review of the various models produced during domain analysis. In particular the ASSET BASE MODEL that identifies features of interest to customers and preliminary allocations of groups of features to particular customers is reviewed. The

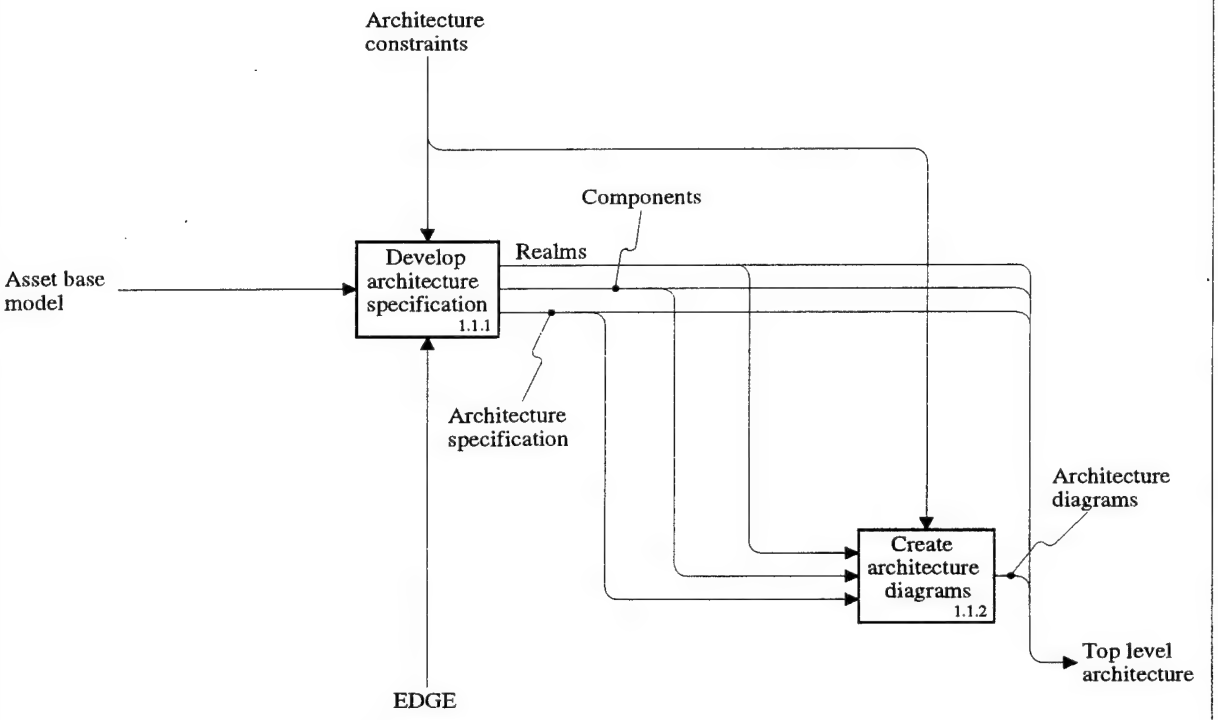



**Exhibit 17.** Develop Top Level Architecture $IDEF_0$ Diagram

architecture constraints are also consulted to see if there are any outstanding issues that may affect the ability of particular customers to successfully build applications around the assets in the asset base. A preliminary attempt is made to list realm and component candidates that group related sets of features found in the ASSET BASE MODEL into an initial implementation strategy for defining these feature sets. Connectivity options among components are represented in the architecture as realm parameters for these components. Layering of features are represented in the realms identified for inclusion in the preliminary architecture.

Existing artifacts (e.g. Ada package implementations of exemplar system code) can often be used to suggest an initial cut at components and their dependence on other components. Interfaces among these artifacts may in turn suggest realm candidates for inclusion in the architecture. Once realm and component candidates have been identified, they can be abstracted and formally recorded using the architecture specification language syntax supported by the EDGE/Ada toolset.

## Results

An ARCHITECTURE SPECIFICATION provides the top-level definition of the architecture. This specification identifies the ***realms*** in the architecture, the ***components*** for each realm, and the realms used by each of these components (called ***realm parameters***).

The ARCHITECTURE SPECIFICATION is supplemented by diagrams of the architecture. These diagrams show a graphical depiction of the architecture including how the architecture is decomposed into realms and how realm components make use of other realms within the architecture. SAAM diagrams (a diagramming method used in the SEI's Software Architecture Analysis

Method [8]) can be used to present a block diagram of the architecture (see Exhibit 21 on page 42). An RLF (Reuse Library Framework) [21] representation of the architecture can also be produced, generated automatically by EDGE/Ada from the ARCHITECTURE SPECIFICATION. The RLF representation of the architecture can be used to support application engineers in their retrieval and composition of assets from the asset base during the *Apply Asset Base* phase.

### Process

As shown in Exhibit 17, the *Develop Top Level Architecture* sub-phase consists of two tasks:

- In the *Develop Architecture Specification* task, domain engineers first map their informal architecture conceptions derived from domain model data to a formal description using the language supported by EDGE/Ada. EDGE/Ada then compiles this specification into the EDGE library. Elements of this architecture specification are used by the EDGE/Ada toolset during later phases of the architecture definition process.
- In the second task, *Create Architecture Diagrams*, domain engineers use documentation or CASE tools to produce diagrams and other informal notes about the architecture as it is being created.

### Sequencing

- The two tasks are usually worked simultaneously. ARCHITECTURE DIAGRAMS can be created from the domain information sources and the resulting figures then converted to ARCHITECTURE SPECIFICATIONS. Or, the ARCHITECTURE SPECIFICATIONS can be produced directly from the information sources and then diagrams can be created to help visualize and summarize the architecture. In either case, the domain engineering staff needs to make sure that the ARCHITECTURE DIAGRAMS are accurate and reflect the contents of the formal ARCHITECTURE SPECIFICATION.
- Neither task can be considered as driving the other. Both of these tasks consider the same raw information sources and workproducts can be produced using these sources independently. However, the team may elect to always perform one task as a lead-in to the other so that the second task can be considered as a derivative of the first. This strategy will lessen the risk of discontinuity between the information recorded during these tasks.
- Adequate attention must be given here before considering other architecture tasks. While there are no hard sequencing constraints on these tasks, it would be a mistake to spend inadequate time in performing the tasks in this sub-phase before going on to do significant work in other areas. Defining a top-level architecture view, even a preliminary one that may change significantly as the asset base is evolving, will help the entire domain engineering team to think in architectural terms and base further development work on a formally recorded architectural foundations.

## 5.1.1 Develop Architecture Specification

Domain modeling has helped the domain engineering team understand what is required of assets that are to be developed. In the *Develop Architecture Specification* task, the framework is created by which these assets are configured, understood and applied. This framework is the ASSET BASE ARCHITECTURE. DAGAR, supported by the EDGE/Ada toolset, gives the domain engineer the means to create the architecture in a machine-processable form. The architecture will then be used to guide asset development and will be the basis for the infrastructure by which the application engineer selects assets for use within a particular application.

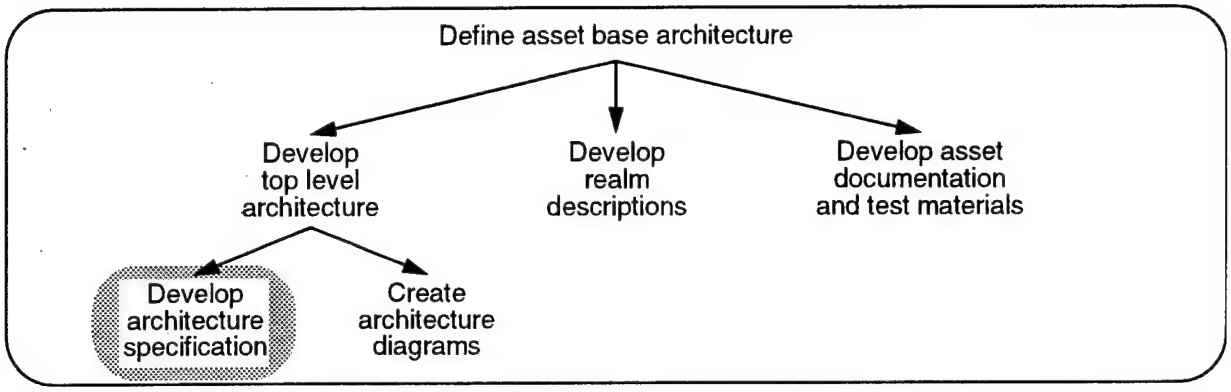



**Exhibit 18.** Develop Architecture Specification Process Tree

The primary purpose of the *Develop Architecture Specification* task is to complete a formal ARCHITECTURE SPECIFICATION that is syntactically correct and that will evolve to include declarations for all of the basic architectural entities that make up the architecture (realms) and a complete list of all of the components that belong to each of these realms. The EDGE/Ada-supported language used to define the architecture is an adaptation of one originally defined by Don Batory in his support of the DSSA project that produced architectural specifications for the avionics domain [1]. As a result of deciding the identities and gross architectural characteristics of the building blocks, the domain engineering staff will be prepared to begin the development of interface specifications for both the realms, and components within each realm. The architecture will also be used as the basis for DAGAR component composition expressions that govern the generation of Ada packages from the architecture.

Avoid the temptation to pay attention at this point to details of the interfaces and functioning of the realms and components identified during this task. The architecture should not be viewed as something that can "fall out" of the effort to complete the realm and component specifications. The DAGAR process is grounded in the need to give adequate attention to up-front architecture development and specification. While the toolset can insure that every part of the asset base architecture and contents are consistent, the architecture itself definitely does not fall-out from other aspects of domain engineering.

## Approach

Developing an architecture specification involves identification of realms, identification of components (including realm dependencies for each of these components), and the production of the ARCHITECTURE SPECIFICATION itself using the EDGE/Ada toolset.

## Workproducts

- REALMS

The list of realms that are candidates for inclusion within the ASSET BASE ARCHITECTURE. This list should include whether each candidate realm was included in the architecture or rejected, and, if rejected, the reason for rejection. This is a working list that is used to support the production of the ARCHITECTURE SPECIFICATION. Identification of realms that were considered, but rejected, for inclusion in the architecture can help to train other domain engineers in learning how to decide what realms ought to make the cut. It may also be useful to go back to this list of realm candidates

```
-----------------
-- File: elpa.e
-----------------
realm
   Coordinates,
   Data,
   Database,
   Dates,
   ELPA,
   ELPA_Math,
   Error_Comp,
   Files,
   Filter,
   Fix_Calc,
   Fix_Math,
   Fixes,
   Matrices,
   Session,
   Vectors;
-- ...
```

```
-----------------------
-- Computation Layer:
-----------------------
   Error_Comp = {G_N_Error_Comp[ELPA_Math, Matrices],
                 ...};

   ELPA_Math = {Verdix_ELPA_Math,
                ...};

-------------------------------------------------------
   Matrices = {Std_2_Matrices [Vectors],
               Std_3_Matrices [Vectors],
               Sym_2_Matrices [Vectors],
               Sym_3_Matrices [Vectors],
               ...};

   Vectors = {Std_2_Vectors [ELPA_Math],
              Std_3_Vectors [ELPA_Math],
              ...};
-- ...
```

**Exhibit 19.** Partial Architecture Specification

later when architecture modifications are under consideration so that possible alternative formulations can be extracted from this list.

■ COMPONENTS

The list of components considered for each realm, along with identification of which were selected or rejected and the rationale for the choice. As is the case for realms, the list of considered components is used to produce the ARCHITECTURE SPECIFICATION. The list may also prove useful for training purposes and to serve as source material when the architecture undergoes modifications as construction of the ASSET BASE ARCHITECTURE proceeds.

■ ARCHITECTURE SPECIFICATION

An ARCHITECTURE SPECIFICATION provides the top-level definition of the architecture. The ARCHITECTURE SPECIFICATION identifies the REALMS in the architecture, the COMPONENTS for each REALM, and the REALMS used by each of these COMPONENTS (called ***realm parameters***). The first two activities generate lists of candidate elements for the realms and components that will be formally named and related within the architecture specification itself. In fact, it is likely that all three of these activities will be performed simultaneously and that various ARCHITECTURE SPECIFICATION drafts will be produced and processed by EDGE/Ada as edited versions of realm and component candidates are pulled from the raw listings.

Two segments of an ARCHITECTURE SPECIFICATION for the ELPA domain produced during the STARS Army demonstration project are shown in Exhibit 19. The left box of the figure declares all of the realms that are part of the specification while the right box shows the declaration of components for four of the realms. The names in square brackets are the realm parameters for each component. For example, the component **`G_N_Error_Comp`** in the **`Error_Comp`** realm is declared to have **`ELPA_Math`** and **`Matrices`** as realm parameters. The ARCHITECTURE SPECIFICATION is processed by the EDGE/Ada toolset to place the top-level architecture definition in the EDGE library.

### When to Start

- Sufficient Domain Analysis workproducts exist. As discussed at the beginning of this section, DAGAR officially begins with the top-level architecture production set of activities and the production of the necessary information resources with which to perform the activities is outside the scope of this guidebook.

- Asset Base customer context understood. The asset base customer context is a vital segment of the information web that should be consulted while domain architecture definition activities are taking place. The asset base should be constructed to support specific customers and customer needs and at the same time have properties that will fit the needs of anticipated customers. In order for architecture definition to be successfully carried out, this customer impact assessment must be available to, and understood by, the domain engineering staff.

### Inputs

- ASSET BASE MODEL. This workproduct contains a collection of domain analysis results that can be used as the basis for architecture formulation. What must be known is how domain features were arranged and provided for in past applications (e.g., in exemplar systems considered during domain analysis activities) and a prescriptive model of how these and other desired domain features can be arranged and implemented to meet the needs of a set of actual and projected asset base customers.

  As suggested in [17], an architecture driven process such as DAGAR needs to be supported by architecture-centric domain models. Thus, it is helpful to know during domain analysis that a process such as DAGAR will be used to apply the results of the analysis during architecture definition. Knowing this will help the domain analysts produce models that will possess an *architecture perspective* that can anticipate some of the essential choices that will eventually be made and presented during architecture development,

### Controls

- ARCHITECTURE CONSTRAINTS. The identification of these constraints is outside the scope of DAGAR. A priori knowledge of the intended application of the DAGAR methods can lead to tailored versions of these constraints that are more easily applicable to decision-making during architecture development.

### Activities

➤ Identify REALMS

In the early stages of architecture development, REALMS represent potential groupings of domain functionality and features into a collection of abstract object and operation interfaces. At this stage, the domain engineer is not yet concerned with making decisions about the precise number or format of these interfaces. However, the placement of such operations and objects into compatible groups and looking for high-level layering of such segmented collections into a series of abstract state machines are both important considerations. Naming these collections and determining a gross interconnection strategy among the various groupings is an important first step in producing the domain architecture. These collections or layers of capability will be fleshed out in the *Develop Realm Descriptions* task.

The primary output of this activity will be a list of realm candidates and a brief narrative description of the scope, level and purpose of each of the realm candidates. Not all of these may be used

in the ARCHITECTURE SPECIFICATION but unused candidates can serve as comparative touchstones as architectural reevaluations is performed in response to asset base implementation activities. The primary information source for this activity is the ASSET BASE MODEL produced during the latter stages of domain analysis. The set of realms will likely be produced in tandem with a set of components as identified in the next activity. Both of these sets provide the raw material from which to build the ARCHITECTURE SPECIFICATION.

➤ Identify COMPONENTS

Each abstraction of domain functionality and feature sets encapsulated as a realm will be realized as one or more implementations. A realm implementation is called a COMPONENT. Since there are many potential ways to achieve the capabilities identified as realm objects and services, there are many potential components that can provide what is offered through a realm interface. This activity seeks to develop candidate lists of alternate implementation strategies and approaches that can deliver the capabilities required of the realm.

The primary way components can differ from one another in a DAGAR architecture is in the number and level of resources that each component expects to use to complete its mission. These supporting resources are typically recognized as realms that belong to the architecture itself and are parameters to the individual components. In addition to the architectural dependencies, a component may need to reference a number of infrastructure or supporting packages that exist to support components at all levels of the architecture. These packages are not part of the architecture as seen by the outside world (e.g., an application engineer wishing to apply the architecture) and may be written as Ada modules.

The purpose of this activity is to assemble a list of component alternatives for each realm and to identify what realm parameters and supporting packages may be necessary to complete the implementation of components. No details are required at this stage, and a tendency to prematurely consider implementation details must be resisted. An iterative strategy where a component alternative set is produced as each realm candidate is identified is one possibility. Another is to first generate the candidate list of realms devoid of any component considerations and then begin the process of component identification.

➤ Create formal ARCHITECTURE SPECIFICATION

As previewed in Exhibit 19, the primary output of this task is a formal architectural description that is processed by EDGE/Ada. In the early stages of architecture development, what is know is a list of realms (the left side of the exhibit) and for each realm, a list of the components that are expected to be built to implement each realm. Each component may have one or more realm parameters (enclosed in square brackets, separated by commas) that declare what architectural resources outside the component's own realm are necessary to complete the implementation of the realm by the component. The right side of Exhibit 19 gives several examples of components and their associated realm parameters.

The domain engineer will use a text editor to prepare the architectural specification (by convention, saved as <filename>.e) and this file is then processed by the EDGE/Ada toolset. Any syntactic errors are reported to the user. The set of realms, and components per realm, are entered into the EDGE library and this stored data will be used to confirm the correctness of realm and component specifications as these are produced in later phases of the DAGAR process. For example, if the architecture specification declares that a particular component requires three distinct realm parameters, later when the domain engineer wishes to "compile" the component specification and body for the component, only those three realms will be allowed in any external resource references made from the component (along with any supporting packages).

The domain engineer can elect to wait until relatively mature realm and component lists have been produced during the other two activities comprising this sub-task; or, EDGE/Ada can be used early and often to prototype a skeletal rendition of the architecture specification as new realms and components are identified as being worthy of serious consideration. Each successive application of EDGE/Ada on the same file will update the EDGE library to include the modified realm and component descriptions.

### When to Stop

- Mature and workable ARCHITECTURE SPECIFICATION exists. The domain engineering team should not expect that the entire architecture can be finally and completely determined after a single pass through the activities of this sub-task. However, a sufficiently stable and complete version of the architecture must be in place before beginning asset base implementation. The other tasks of the *Architect Asset Base* task, in particular the *Develop Realm Descriptions* task, can be begun before the architecture has completely stabilized. Work in these areas can in fact point out weaknesses in the architecture that need to be addressed before the later phases of DAGAR get underway.

  The *Develop Architecture Specification* task can stop when the set of realms and components is understood well enough that component specification and component body development can begin.

### Guidelines

- Iterate back to Asset Base Modeling as necessary. If the domain engineering staff has built the right kinds of models, and gathered the right sort of information within models, it will probably be straight-forward to transition from those models to the architecture for an asset base. If in fact it turns out that this transition experience is difficult, the domain engineering team should consider revisiting its ASSET BASE MODEL development processes with the aim of identifying process improvements and improving the ASSET BASE MODEL.

- Frequent consultation of domain information sources should not be needed. The ASSET BASE MODEL is the primary information sources for architectural development. Frequent trips back to domain information sources suggest the need for an improved process and better model review activities.

  There should be a built-in mechanism within an organization's pursuit of an ASSET BASE ARCHITECTURE so that when a disconnect between ASSET BASE MODEL applicability and usefulness is detected, the domain engineering staff is able to go back, modify their procedures and feature "sensitivity" and produce better models that support the development of the ASSET BASE. However, it would be a mistake to attempt to model everything. Pointers back to the domain information sources can be provided and used to fill in details as necessary. But such details are more likely relevant during the later implementation phase of DAGAR rather than in the architecture definition phase.

- Where possible, directly connect features to architecture. The most common connections from features to architecture should be visible as feature model support for declarations appearing in REALM SPECIFICATIONS and for elaborations of holes, and the presence of particular realm parameters, within COMPONENTS.

  Features should generally be citable for the grouping of types and operations within REALMS and for the variability supported within the architecture for providing adaptations of these REALMS by COMPONENTS. Support for the creation of major REALMS and COMPONENT implementations of these REALMS may also be found in the domain stakeholder model where

certain types of desired configurability have been recorded. Innovative features can be used to support certain architectural decisions, but there should also be collateral stakeholder support for building these decisions into the ASSET BASE ARCHITECTURE. In any event, the structures presented through the REALM SPECIFICATIONS, and the variability indicated by the presence of multiple COMPONENTS with unique connections to other REALMS within the ASSET BASE ARCHITECTURE, should be traceable to feature model elements as architecture definition is carried to its conclusion.

- Start defining the TOP LEVEL ARCHITECTURE DESCRIPTION early. It is critical that this task not be delayed. If necessary, the best exemplar system architecture should be evaluated for use as a strawman ASSET BASE ARCHITECTURE and expressed as a DAGAR ARCHITECTURE SPECIFICATION.

## 5.1.2 Create Architecture Diagrams

The task described in the previous subsection was aimed at formalizing what can be said about the top-level architecture as architecture definition begins. But such a formal description can be hard to visualize and communicate to someone other than the domain engineers who helped develop the formal architecture specification. As such, descriptive material that displays the architecture in an expressive, visual manner is desired.

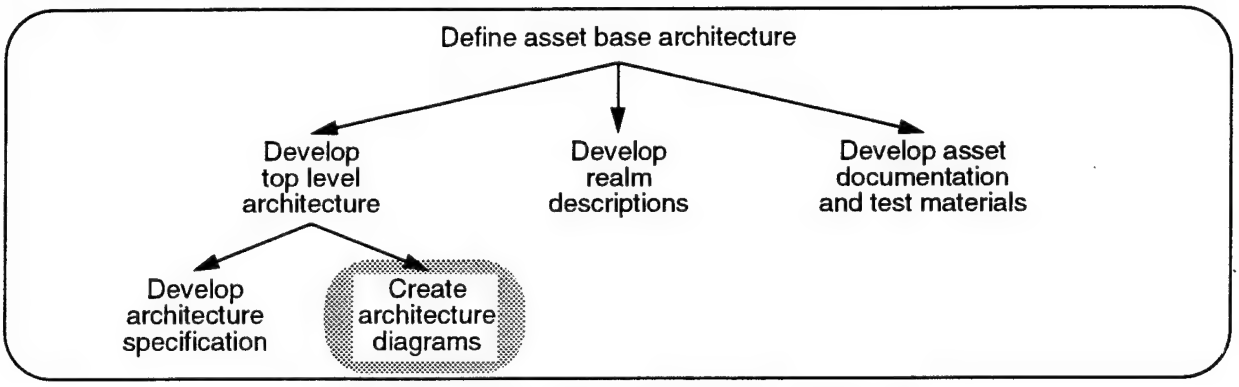



**Exhibit 20.** Create Architecture Diagrams Process Tree

The primary purpose of the *Create Architecture Diagrams* task is to provide such a visual and communicable rendition of the architecture. At present, there is no built-in support in EDGE/Ada for automatically generating this material directly from the architecture. Hence, a domain engineer must be tasked with producing it manually from the formal architecture.

With such a manual conversion process, there is a danger that the two forms of architecture rendition will become unsynchronized with respect to each other. Consequently, a staff member needs to be assigned responsibility to periodically check to make sure that the forms of the architecture are both describing the same architecture and to re-draw the diagrams as necessary to achieve compatibility.

### Approach

The ARCHITECTURE SPECIFICATION is supplemented by diagrams of the architecture. These ARCHITECTURE DIAGRAMS show a graphical depiction of the architecture including how the architecture is decomposed into REALMS and how realm COMPONENTS make use of other realms within the architecture. SAAM diagrams (a diagramming method used in the SEI's Software

Architecture Analysis Method [8]) can be used to present a block diagram of the architecture. An RLF (Reuse Library Framework) [21] representation of the architecture can also be produced, generated automatically by EDGE/Ada from the ARCHITECTURE SPECIFICATION. The RLF representation of the architecture can be used to support application engineers in their retrieval and composition of assets from the asset base during the *Apply Asset Base* phase

The EDGE/Ada toolset can also provide its own tree-like description of the set of realms and components. This tree view is used during system composition to show the application engineer the set of available component choices. Interestingly, this view is not yet presented in EDGE/Ada as a visualization mechanism to be used during architecture development. Consideration is now being given to making this mechanism useful during architecture development.

When summary diagrams are being produced by hand, they can either be done before committing the architecture in a formal specification (the ARCHITECTURE SPECIFICATION file) and so be used to guide the development of the formal architectural description; or, they can be produced after the formal architecture has been processed by EDGE/Ada. Either of these approaches is acceptable and which is preferable depends on the visualization skills and needs of the domain engineering team.

## Workproducts

- ARCHITECTURE DIAGRAMS

Exhibit 21 gives an example of an architectural representation that the Army STARS Demonstration project found useful as a communication mechanism in talking about the Emitter Location Processing and Analysis (ELPA) domain architecture. This diagram was produced using FrameMaker. For a large architecture, several such pictures may be required. The small, solid color boxes in the picture represent realms and the lines between boxes represent the potential for components within the realm having the realms at the level(s) below it as realm parameters. Thus, the lines from the **`Fix_Calculations`** realm to **`Filtering`**, **`Fix_Math`** and **`Coordinate_Transformations`** indicate that at least one component in **`Fix_Calculations`** declares each of these as a realm parameter

The realms within an enclosing rectangle are roughly at the same level of generality in that they deal with data at the same granular level. Exhibit 21 shows three such levels: Services Layer, Computation Layer and the Data Access Layer.

In the ELPA architecture, there was only one planned communication mechanism, that of procedure invocation with values returned either as function return values or through procedure parameters. As such simple lines drawn between realms indicate that the definitions of services in one realm may invoke the procedural and functional interfaces of services in another realm in order to complete the required computation. If several communication mechanisms are planned for the architecture (e.g., tasks, remote procedure call, etc.) then several diagrams may be necessary for the same segments of the architecture to represent the issues and alternatives being addressed through the separate communication mechanisms.

Note that these diagrams are not meant as substitutes for design diagrams that might be used in the development of the components themselves. Such diagrams are at a much finer level of detail and can prove useful during component implementation. The diagrams here are architectural level diagrams meant to supplement and support the formal language used in the ARCHITECTURE SPECIFICATION.

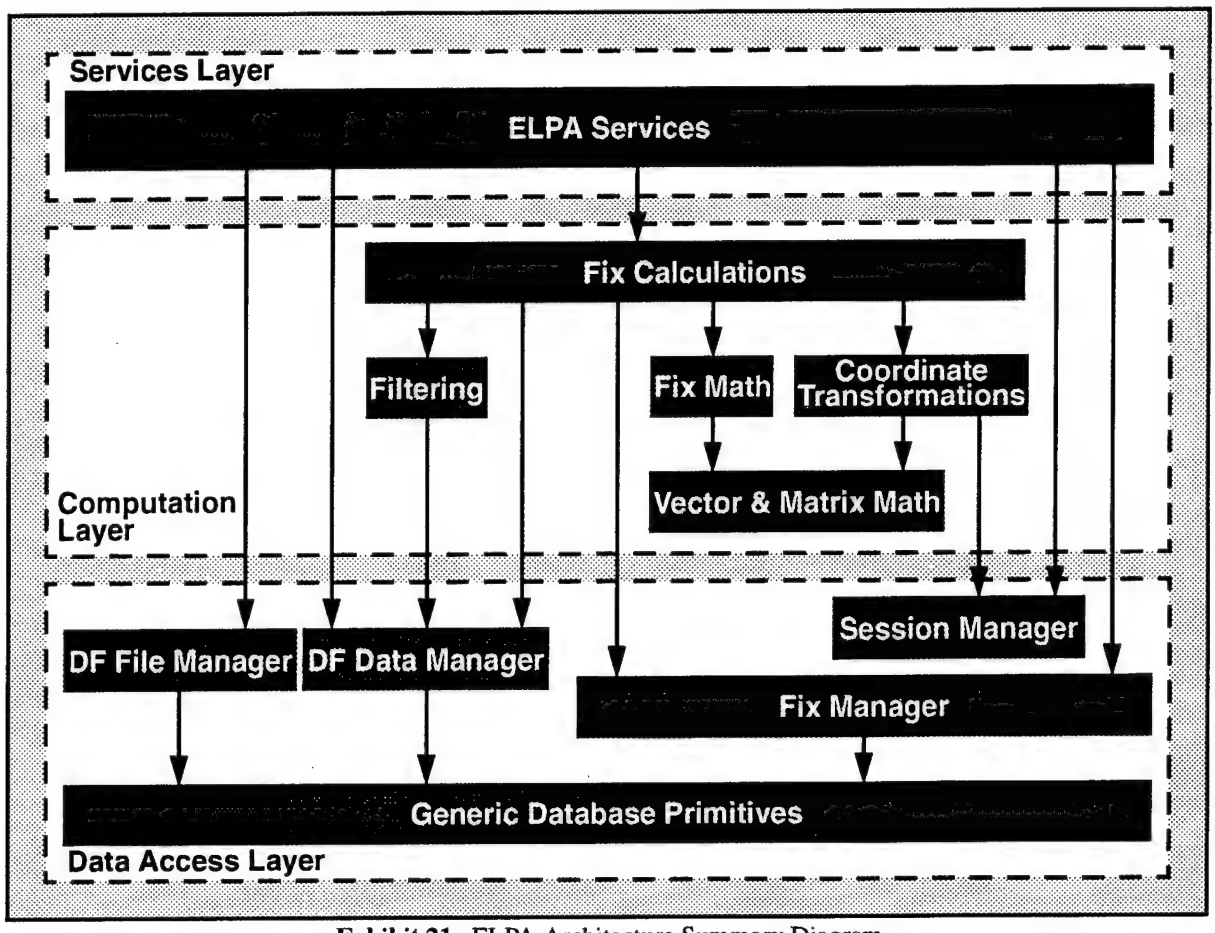



**Exhibit 21.** ELPA Architecture Summary Diagram

## When to Start

The production of architecture diagrams can begin at the same time as the *Develop Top Level Architecture* task begins. Thus these two tasks have identical starting conditions.

## Inputs

- REALMS. Realms are represented as boxes in the architecture diagrams produced during this phase.
- COMPONENTS. Components within realms have various realm parameters. The existence of a component in a realm with a realm parameter will generally lead to a line being drawn in the diagram between the two realms.
- ARCHITECTURE SPECIFICATION. The Architecture Diagrams are supposed to provide visualization aid and support for the domain architecture. Although early versions of these diagrams can be produced before the formal ARCHITECTURE SPECIFICATION is completed, there must be final agreement between baselined versions of the formal architecture and the architecture diagrams. As such, the ARCHITECTURE SPECIFICATION can be considered an input to this sub-task.

## Controls

- ARCHITECTURE CONSTRAINTS. The production of the architecture diagrams is "controlled" by the same considerations that are applicable to the production of the formal ARCHITECTURE SPECIFICATION.

## Activities

➤ Produce Architecture Diagrams.

The methods used to produce the diagrams are highly dependent on the tool used to create the diagrams and whether the domain engineering team is using this activity as a lead-in or follow-on activity to the production of the formal architecture. Briefly, the team should use whatever approach they feel most comfortable with and which is compatible with the drawing or CASE tool employed to actually make the drawings.

The team should be aware of what part of the architecture comes first in the architecture definition effort: diagrams or formal architecture. It may also be possible to produce these two views of the architecture as a tightly bound coordinated activity so that as understanding of, and detail in, the architecture is added, the two views are modified accordingly at the same time.

➤ Verify consistency between Architecture Diagrams and ARCHITECTURE SPECIFICATION

No matter how the architecture diagrams are produced, their content must be evaluated and compared to the information contained in the formal architecture specification. Since the formal specification is what is enforced by the EDGE/Ada toolset, this version of the architecture is the official one. Even if the diagrams are produced in advance of the formal architecture, architecture editing with EDGE/Ada may result in changes that need to be reflected back in the diagrams. If the production of the diagrams lags behind the formal architecture, care must be taken that the elements drawn in the diagrams and their interconnection accurately reflect the architecture itself.

This activity will need to be repeated as later phases of the DAGAR life-cycle lead to architectural modifications. Most likely these changes will be handled directly through EDGE/Ada and so the diagrams will then need to be edited to reflect the changes as they are made. A better solution would be to have EDGE/Ada support some level of diagram generation from the formal specification. Such an extension to EDGE/Ada may be added in a future release.

## When to Stop

The production of architecture diagrams can stop at the same time as the *Develop top level architecture* task stops.

## Guidelines

- Consider a three-layered TOP LEVEL ARCHITECTURE DESCRIPTION. Both the ELPA ASSET BASE developed by the Demonstration Project and the avionics ASSET BASE addressed by the Loral DSSA team show that a basic three-level layered architecture provides an effective starting point for a DAGAR ASSET BASE ARCHITECTURE. This approach may be applicable to a wide variety of domains.

  As shown in Exhibit 21 on page 42 for ELPA, and Exhibit 22 for a web site domain being built on top of the OpenRLF, a three-layered architecture comprised of:

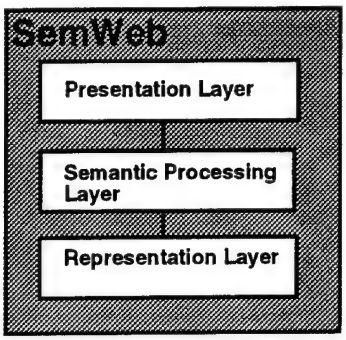



**Exhibit 22.** SemWeb Architecture Summary Diagram

— an external interface layer through which client applications connect to basic ASSET BASE services,

— a middle core computational layer within which basic domain abstractions are processed, and

— a bottom data access layer through which client data is accessed as necessary to feed the computational layer

has proven to be an effective organizational technique, especially in the early phases of architectural development. There eventually will be several internal REALMS that subdivide these baseline layers (especially for the core and data access layers). As shown in Exhibit 21 for ELPA, COMPONENTS for REALMS within a layer will in general require access to REALMS at a lower level, but these access points should be controlled to avoid needless complexity or unnecessary REALM-to-REALM interactions. If a domain is determined to require more than three layers (e.g., it makes sense to stratify the middle layer to support information hiding concerns), access to REALMS more than one level removed from the current level should be questioned. An exception to this recommended restriction is access to REALMS in the data access layer to provide access to persistent data within the client application.

- Decide on what definition activity comes first and stick to this decision. It is up to the team performing architecture definition to decide which task comes first (formal architecture or architecture diagrams) or whether a simultaneous evolution approach is preferable. But the team should always be aware of what strategy is being followed and its procedures should consistently mirror this strategy.

## 5.2 Develop Realm Descriptions

Realm specification can begin as soon as enough confidence in the early phases of architecture definition reaches the level that realm identities are stable enough that engineers can identify the services and objects to be made available through the realm interface. For some of the realms in the architecture, full details about these services may not be available until much later in the DAGAR process when the impact of implementation choices begins to ripple back to the architecture itself. However, it would be a mistake to postpone realm definition activity until full knowledge about what a realm needs to provide is available. It is better to treat the production of realm descriptions as a prototyping activity where initial versions of realm descriptions for all of the realms initially identified in the ARCHITECTURE SPECIFICATION are created and processed via the EDGE/Ada toolset. As details are added, or specification elements change, the corresponding REALM DESCRIPTION can be edited and re-processed. While the ARCHITECTURE SPECIFICATION shows the major elements that make up the architecture in the form of realms, what is contained within each of the elements is external to this specification. This detail is what is communicated in each REALM DESCRIPTION.

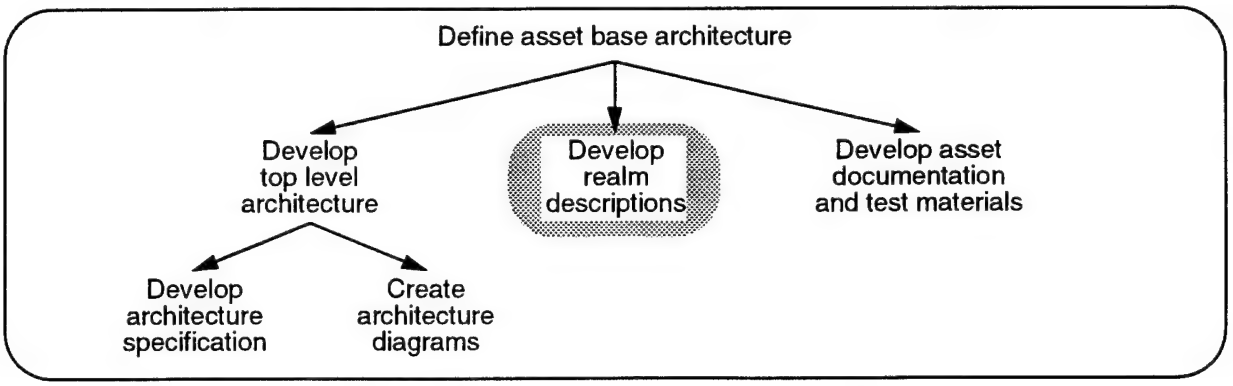



**Exhibit 23.** Develop Realm Descriptions Process Tree

The primary purpose of the *Develop Realm Descriptions* task is to use the facilities of EDGE/Ada to elaborate to the extent possible what each realm in the architecture is expected to provide to the other realms in the architecture. The Realm Description language supported by EDGE/Ada is based on the Ada language and the content and scope of a REALM DESCRIPTION will be similar to what is seen in an Ada package specification. As such, a list of operations is given, including the parameter profile expected for each operation, along with a set of basic data type abstractions that define the kind of objects that the realm expects to process and produce.

Since realm descriptions will wind up looking a lot like Ada package specifications, a danger exists that as engineers will begin to view them as synonymous. There is one major difference in how Realm Descriptions are constructed when compared to Ada package specifications. Realm descriptions will contain various "holes" where details concerning how a capability to be offered through the realm interface is actually going to be provided. Each such hole amounts to an implementation detail that the realm delegates to a component to specify. Because a component in all cases is a realm implementation, it is a component's responsibility to provide these implementation details. Every hole in a realm specification corresponds to a hole filler clause in a component specification. If the hole represents an optional part of the realm, then the hole filler may be empty.

In designing realms and completing their descriptions, a key challenge will be to anticipate places where variations on how some piece of interface capability can be completed. A hole is left in the REALM DESCRIPTION that can accommodate a component's ability to provide different choices

and therefore different ways to fulfill these variations. Training in the design of software abstractions that support this kind of extended configurability is required. A person used to designing standard Ada package abstractions may not easily shed the systems mindset that such package design experience imprints on the engineer. Knowing when (and how) to leave something out is just as important as knowing when to leave something in.

## Approach

Realms are described in REALM DESCRIPTIONS written in an extended Ada notation that includes holes that can be filled in by components. A partial REALM DESCRIPTION for a realm providing matrix processing services is shown in Exhibit 24. The explicit areas in a realm that components can use to tailor the implementation of the realm are indicated by identifiers enclosed in square brackets (e.g. **[Matrix_Index_Declaration]** and **[Matrix_Declaration]**). The identifiers enclosed in double curly braces (**{{}}**) provide additional substitution points that are supported by EDGE/Ada on an architecture-wide basis, rather than for individual components. The EDGE/Ada toolset processes the REALM DESCRIPTION to place it in the EDGE library.

Knowing the major modules that make up the domain architecture marks the beginning of the architecture development process. The next step is to determine what features and capabilities are encapsulated in each of the realms. The major source of information used to make this determination is once again the ASSET BASE MODEL. Exemplar systems used in the production of this model may in fact provide packaging and interface information that will be helpful in formulating draft versions of the REALM DESCRIPTIONS. Work on producing the REALM DESCRIPTIONS should not begin until the ARCHITECTURE SPECIFICATION as a whole has stabilized and undergone signif-

```
-------------------------------------------------------------------------
-- File: matrices.r
--
-- This is a realm  that defines operations between 2x2 symmetrical
matrices;
-- 3x3 symmetrical matrices; 2x2 non-symmetrical matrices;
-- 3x3 non-symmetrical matrices;
--
-------------------------------------------------------------------------

with {{ELPA_Support}}; use {{ELPA_Support}};
with {{Base_Support}}; use {{Base_Support}};

-------------------------------------------------------------------------
$realm package Matrices is

   [Matrix_Index_Declaration]

   [Matrix_Declaration]
-- ...
function "*" (Left, Right : in Matrix)
      return Matrix;

function "*" (Left  : in Matrix;
                  Right : in [Vector_Declaration])
      return [Vector_Declaration];

function "*" (Left  : in [Vector_Declaration];
                  Right : in Matrix)
      return [Vector_Declaration];
-- ...
```

**Exhibit 24.** Partial Matrix Processing Services Realm Description

icant internal review. Beginning work on realm interface definitions too soon raises the potential of unnecessary or unused work.

## Workproducts

- REALM DESCRIPTIONS

As illustrated in Exhibit 24, the basic format of a REALM DESCRIPTION has the appearance of an Ada package specification. The most important difference to note in comparison to Ada package specifications are the uses of identifiers enclosed in a pair of single brackets (`[]`) or double curly braces (`{{}}`) that are used to mark the appearance of substitution points (or holes) that other parts of the architecture definition have the opportunity of providing values for. The example shows three such holes that components in the realm are expected to define: one for a matrix index type, one for a matrix description that must include the Ada definition for a **`Matrix`** type and one for a Vector type identified by the identifier **`Vector_Description`**. Components will provide definitions for each of these named substitution points in the form of Ada code fragments that will allow the Ada compilation of the REALM DESCRIPTION after the substitutions are made (see Exhibit 32, "Complete Matrix Component Specification," on page 71). The content of these substitutions will vary from component to component.

In addition to the component-specific holes, there are architecture-wide substitution points (delimited by a double pair of braces) that EDGE/Ada will provide values for. One such hole is identified as **`ELPA_Support`** and, as can be seen from the surrounding syntax, names an Ada package that is "withed" to provide the definition of auxiliary services needed to support the rest of the REALM DESCRIPTION (note that there are no statements shown in Exhibit 24 that actually use definitions that are contained in this support package).

## When to Start

- ARCHITECTURE SPECIFICATION sufficiently complete. While the ARCHITECTURE SPECIFICATION does not need to have been completely fleshed out, it must be in a state that reasonable certainty exists regarding the identity of some key realms in the architecture and possible connections between components in these realms and other realms. REALM DESCRIPTIONS begin to add a layer of detail into the architecture that can require some significant design effort. It would not be economical to have the expenditure of this effort not lead to results that can be carried forward into the next phases of the domain engineering project.

- Coordination with Asset Implementation effort determined. In order for the *Develop realm descriptions* task to lead to compilable Ada, there must be at least one component available for each realm to provide hole fillers. If these components themselves rely on their own realm parameters to complete the definition of any hole filler, then at least one component for each of the realm parameters must also exist. Thus, if it is desired that early-on in the realm definition effort, a translation to Ada (using the services provided by EDGE/Ada) is desired, sufficient components must have been added, at least down to the COMPONENT SPECIFICATION level, to enable the production of Ada code. The domain engineering project needs to work out what it will do in the face of such dependencies and plan actions accordingly.

## Inputs

- ARCHITECTURE SPECIFICATION. Realms only exist in the context of the architecture and REALM DESCRIPTIONS can only be processed successfully if their connectivity to the architecture is adequately documented by the current ARCHITECTURE SPECIFICATION.

- ASSET BASE MODEL. This model provides information about exemplar systems as well as customer needs and wants regarding asset base services. Knowledge of both of these areas will be useful in making architectural decisions including the structure and content of REALM DESCRIPTIONS.

## Controls

- ARCHITECTURE CONSTRAINTS. The production of the REALM DESCRIPTIONS is "controlled" by the same considerations that are applicable to the production of the formal ARCHITECTURE SPECIFICATION. How features and capabilities are arranged into layers, and how these layers are interconnected to one another, may be influenced strongly by many factors. The purpose of internal and external ARCHITECTURE CONSTRAINTS, at least as formulated by ODM, is to record and weigh such factors.

## Activities

➤ Write each REALM DESCRIPTION

The actual task of completing a REALM DESCRIPTION is not unlike that of completing an Ada package specification and so design techniques practiced within the organization can be applied. The real problem to be addressed is that each realm represents not just one package but potentially many different packages with different behaviors and performance characteristics. The designer must anticipate and accommodate the variability that asset base customers may find useful at each level in the architecture and leave room in the definition (through the holes as discussed above) to provide this variability.

➤ Verify each REALM DESCRIPTION

While EDGE/Ada can check the identity of the realm (e.g., the realm name as stated in the Realm Description) with respect to the EDGE library, much of the content of a REALM DESCRIPTION is Ada code that is not parsed by the EDGE/Ada toolset. EDGE/Ada leaves the processing of these details to the Ada compiler. In order that an Ada compiler pass over the REALM DESCRIPTION can be successfully completed, there must be substitutions made for each of the holes in the description. Such substitutions require that at least skeletal versions of COMPONENT SPECIFICATIONS exist in the architecture. These skeletons can be written here for realm verification purposes or could be developed as an early activity within the *Develop Component Specifications* sub-task.

## When to Stop

- Asset Base Architecture draft complete. The asset base architecture consists of both the ARCHITECTURE SPECIFICATION and a collection of REALM DESCRIPTIONS. Both of these products must be sufficiently mature so that development activities within the *Implement Asset Base* phase of DAGAR can productively take place. Work on the architecture can be resumed as conditions warrant but the initial set of activities performed here must produce sufficiently many and complete REALM DESCRIPTIONS for asset base implementation activities to begin.

- Checkpoint with Asset Base Implementation effort reached. As part of the overall DAGAR-based domain engineering effort, production of the architecture and in particular the completion of Realm Descriptions can be viewed as an asset base design activity that is coordinated with a set of implementation activities. As desired, a set of checkpoints can be established at which major effort on the project is transitioned between the two phases with perhaps alter-

nate checkpoints causing alternate emphasis being placed on design and implementation activities. When such a checkpoint is reached when the end of major period of design activity is indicated, a transition to implementation effort, using the latest architecture results, can occur.

## Guidelines

- Consider evolutionary development of realms. There are many opportunities with DAGAR to adopt an iterative, evolutionary approach. The production of REALM DESCRIPTIONS, in conjunction with the necessary component material to evaluate the adequacy and correctness of these materials, is a natural process that helps insure that sufficient baseline material is in place to permit testing. Synchronizing realm definition activities with evolution of the architecture itself is also highly recommended so that neither activity produces material that needs to be discarded due to lack of compatibility.

- Use existing exemplar package interfaces as examples. While raw source code is not likely to be directly relevant, a small amount of reverse engineering of exemplar work products that addresses the issues of packaging of services, and communication paths among core sets of services, can be used to create draft realm designs and candidate components that have analogous connections to other realms. While it would be preferable to have one or more exemplars that have Ada implementations, any implementation for which adequate design documentation exists, or can easily be recovered, can be extremely useful in the early stages of the *Develop realm descriptions* task.

- Learn and apply the EDGE/Ada toolset to its fullest extent. The EDGE/Ada toolset was developed to be used in conjunction with the organization's chosen Ada compiler. Ada-specific details are checked by an application of the compiler to the REALM DESCRIPTION after the holes contained in the description have been replaced by component-defined Ada code fragments. EDGE/Ada is used to identify which components are to be used to provide the substitution values for holes in a description and the Ada compiler can be invoked automatically to check the result after the actual Ada code is generated that merges the REALM DESCRIPTION with the information supplied through the COMPONENT SPECIFICATION. Army STARS demonstration project experiences suggest that significant attention should be given to training the domain engineering staff in how to merge what they know from past usage of the Ada language tools with a complete understanding of the capabilities provided through EDGE/Ada.

# 5.3 Develop Architecture Documentation and Test Materials

Within the *Define asset base architecture* phase of DAGAR, domain engineers are specifying the core structure of the asset base. It is important that quality assurance and documentation activities at the architectural level be carried out in conjunction with architecture development. ARCHITECTURE DOCUMENTATION AND TEST MATERIALS will form the basis for the component based materials that will be produced during *Develop asset documentation and test materials* (see Section 6.2.3, "Develop Asset Documentation and Test Materials"). Documentation materials can be produced in template forms that will be elaborated as components are completed. Black box test cases and plans that identify testing concerns at the realm interface level can be developed that are applied to all components as they are completed. Test materials can be produced that indicate the extent to which the variability supposedly implemented in the various component implementations of a realm is actually achieved.

Documentation and test materials supplement the realms and later, the components, to provide the application engineer with a complete package of supportive material about the assets that can be obtained from the asset base. Documentation at the architecture level can include documentation about the overall architecture, and the realms and components within the architecture. Test materials can include test plans, test cases, and test harnesses. Documentation and test materials should first be developed during definition of the asset base architecture and then extended during implementation of the assets.

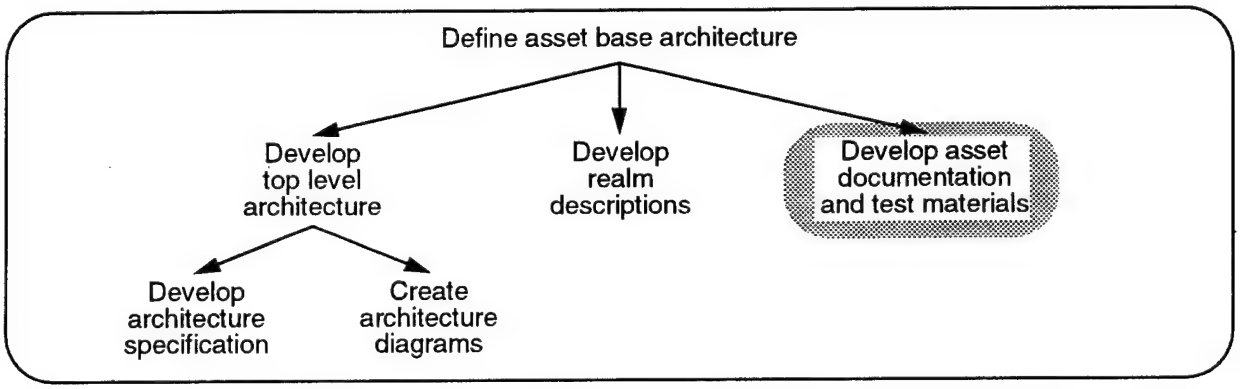



**Exhibit 25.** Develop Architecture Documentation and Test Materials Process Tree

The primary purpose of the *Develop Architecture Documentation and Test Materials* task is to produce the material as described in the name of the task. These materials are produced for the architecture itself, as documented in the ARCHITECTURE SPECIFICATION, for every realm, and for every component. They are used during asset development to understand the intended scope and capabilities of all of the architectural elements and to verify correctness and performance of the realms and components

## Approach

ARCHITECTURE DOCUMENTATION and TEST MATERIALS supplement the realms and components to provide the application engineer with a complete package of supportive material about the ASSETS available from the ASSET BASE. Documentation at the architecture level can include documentation about the overall ASSET BASE ARCHITECTURE and documentation about the realms and components within the ASSET BASE ARCHITECTURE. TEST MATERIALS can include test plans, test cases, and test harnesses. ARCHITECTURE DOCUMENTATION and TEST MATERIALS first devel-

oped during the *Define Asset Base Architecture* phase will be extended during the *Implement Asset Base* phase.

Even during the early phases of domain architecture definition, attention must be given to anticipating the information needs of both the team members responsible for asset implementation as well as the application engineer customers of asset base. Materials developed here should be viewed as seed materials for the final documentation and test products that will be delivered to the application engineer along with the assets themselves. Some of the information first developed and recorded during this task will be packaged for use as part of the ASSET BASE INFRASTRUCTURE.

A primary source of both documentation and test materials is legacy system artifacts. Even though the domain architecture may be the source of a substantially new implementation of application functionality, this new implementation may be used in application circumstances that have been largely unchanged from those surrounding earlier versions of the application. As such, useful material for documentation and testing purposes may already exist among the domain information sources gathered during descriptive domain modeling. Where significantly new functionality is added, or innovative algorithms are designed, the corresponding test and documentation material will have to be developed by the engineer. The usual development methods for such material that is in place within the organization can be applied in producing this new material.

## Workproducts

- ARCHITECTURE DOCUMENTATION

Documentation should be produced in formats that are already in place within the development organization. Supporting material meant for use by application engineers in widely differing usage contexts should actually be made available in a number of different formats. Many documentation development systems allow for the export of material in a number of standard formats that can then be imported to, and accessed from, tools available to the application engineer. Architecture documentation will primarily be concerned with the architecture itself as defined in the Architecture Specification and with documenting the functionality, behavior and connectivity options for each of the realms of the architecture. In addition, skeletal forms of component documentation will also be developed in this task.

- TEST MATERIALS

Test materials include test plans, test cases, test harnesses, expected test results and test result comparers to compare actual results to expected results. Test harnesses will often take the form of special components that are developed as part of the architecture. Such components can be allocated either to existing realms or be placed within special test realms inserted within the architecture to explicitly support asset base testing. Such test realms are not meant to contain components expected to be of use to an application engineering customer except to support other segments of the architecture. The initial identification and positioning of these realms within the architecture should be planned as a necessary and vital part of *Architect Asset Base.* Further elaboration of these realms and the specialized test components that belong to them will occur during the *Develop Asset Documentation and Test Materials.*

## When to Start

- Start of *Define Asset Base Architecture*. This task is best approached as an on-going task as

architecture development proceeds. So even when the *Develop Top Level Architecture* task first begins, work on production of the necessary supporting materials can get started. Supporting materials can then be gradually expanded as more intensive realm development gets underway.

## Inputs

- ARCHITECTURE SPECIFICATION. From an architectural perspective, this product is the key thing that is being documented and must be verified. It should be viewed as the primary entity for which supporting materials are being developed during this task.
- REALM DESCRIPTIONS. REALM DESCRIPTIONS present the interface elements available for each realm. As such, these elements must be reflected in all of the test plans that are begun during this task and each element must be suitably documented so that it can be effectively implemented and invoked.
- ASSET BASE MODEL. This model provides information about exemplar systems as well as customer needs and wants regarding asset base services. Knowledge of both of these areas will be useful in documenting architectural decisions including the structure and content of REALM DESCRIPTIONS. Test cases that verify that customer needs and expectations are met can be framed using the content of this model.
- ASSET BASE DOSSIER. The ASSET BASE DOSSIER is the full compendium of all exemplar material that might be of use in constructing the asset base. In particular, exemplar documentation and test products will be available in the dossier. These can serve as the basis, or at least as a touchstone for comparison, for the creation of analogous architectural products.

## Controls

- ARCHITECTURE CONSTRAINTS. The production of documentation and test materials is controlled by the same considerations that are applicable to the production of the formal ARCHITECTURE SPECIFICATION and REALM DESCRIPTIONS. How features and capabilities are arranged into layers, and how these layers are interconnected to one another, may be influenced strongly by many factors. The purpose of internal and external ARCHITECTURE CONSTRAINTS is to record and weigh such factors. These factors will also apply to how test plans and documentation are created, and how they are related to the architectural elements that they support.

## Activities

Documentation and testing should be done hand-in-hand with the development of the asset base architecture itself. The purpose of this task in DAGAR is to highlight its importance in producing a high-quality architecture. At this time, the EDGE/Ada toolset does not yet provide any enactment mechanisms or work product support for the creation of either documentation or test materials. These must be created and stored using the organization's available software engineering environment services.

➤ Begin Architecture Documentation.

In all aspects of DAGAR-based domain engineering, the domain architecture is the key to producing usable and understandable results. Therefore documentation activities should begin with a careful consideration of the current ARCHITECTURE SPECIFICATION along with the associated architecture diagrams. These diagrams are in fact the first form of architecture documentation. As

necessary, explanatory material is added to document the information flow through the architecture and how the various components within the realms direct or re-direct this information flow.

Holes in Realm Descriptions provide opportunity for architecture variation and the effect and purpose of such holes need to be defined so that component implementation can proceed accordingly. Documentation at the component level can be viewed as describing *how* these holes were filled and the opportunities realized. If component development reveals flaws in the architectural design, a feedback loop in the DAGAR process will provide the opportunity to correct these flaws and the corresponding architectural level documentation must be corrected as well. Since documentation will primarily exist in template form using the locally available documentation mechanisms, engineers will use documentation processing tools to open the templates, manually insert the component-dependent elaboration of the template, and the save a copy as the actual document output of the associated implementation activity.

➤ Review ASSET BASE DOSSIER for possible source material.

While DAGAR domain engineering is based on particular specification and implementation languages supported by the EDGE/Ada toolset, detailed approaches and implementation concepts will often be suggested by previously implemented systems. If these systems were produced using accepted development practices, there will be significant supporting material that was saved as part of the legacy materials from these efforts. Just as the design and implementation of parts of these systems can be used to pattern component designs and implementation, the supporting materials can also be used. If the legacy documentation exists in an electronic form compatible with that being used to support DAGAR, it can be processed electronically and provide valuable starter and supplementary material. Similarly, legacy testing materials (especially test plans and test data) can be used as stepping stones to producing testing material at the architectural level. This material is later refined and extended during component implementation.

➤ Begin and extend architecture testing framework.

Testing is vital to the success of any implementation endeavor. As such, testing must be considered early in the architecture development effort. Most DAGAR-produced asset bases will include at least one, and perhaps several, realms in which the presence of components solely designed to support the testing of other parts of the architecture are indicated. Testing activity, and the production of testing materials, should be focused on the production of these components and creating supporting material (such as test plans and data) for their use. Early attention should be paid to the design of such components and how they can effectively be used to execute testing scenarios based on test data developed during this task. Production of new testing materials should stress effective use of these testing parts and provide data that supports understanding how components compare to one another when accessed from these test components. Individual components can be expected to have certain desired properties and plans and test cases to demonstrate these properties must be developed along with the components themselves.

### When to Stop

- Architecture documentation sufficiently complete. Stopping the documentation process will coincide with the stopping of the architecture development process. If these activities are synchronized with one another, documentation has been proceeding in conjunction with architecture development. If schedules or staffing concerns have led to these activities being de-coupled, attention must be given to reviewing documentation status periodically to insure that and disconnects and inadequacies are addressed.

- Testing materials in place to support asset implementation and application. Asset production

processes will regulate the application of test materials internally to asset development. Testing occurs on a continuous (or at least periodic basis) and materials must be in place to support this testing. The domain engineer must also work to produce testing materials of use at application development and integration time. This means that some understanding of expected usage contexts is required to prepare appropriate testing materials. The *Develop Architecture Documentation and Test Materials* task must continue until sufficient material exists for asset base implementation and the foreseen usage contexts are supported by test materials that will be useful in these contexts. Since later architectural modifications may occur, it may be necessary to return to the task to expand the test materials accordingly.

## Guidelines

- Follow incremental pattern. The production of supporting materials should follow the same pattern as that followed during the production of the ARCHITECTURE SPECIFICATION and REALM DESCRIPTIONS. Echoing the advice given in these task descriptions, the domain engineer should plan the construction of the asset base so that a thread through the asset base is produced as early as possible and that additional layers and more completely functional components are considered as architecture understanding increases. As these initial realms and components are created, the supporting material for these components should be produced along with them rather than be postponed until the end of the creation activity. Once basic materials exist, they can be refined and extended according to the changes made to the architecture itself.

- Ensure sufficient staff communication. While a small project may be run with only a small number of engineers performing all activities in conjunction with architecture development and asset base implementation, for larger projects, staff size increases can present additional problems that need to be worked out. One of these problems is that staff members will often have specialized duties and expertise. Careful hand-offs must be arranged as these duties overlap. The production of documentation and testing materials is often allocated to specialists in these areas and so as development of realm and component drafts and versions occurs, these products must be handed over to those with expertise in documentation and testing.

  Communication among staff members needs to be facilitated and interim documents such as notes, trouble reports, informal test data, etc. needs to be made available as raw material for the *Develop Architecture Documentation and Test Materials* task. EDGE/Ada as currently configured does not offer significant support in this area, so other development platform services need to be selected and applied in support of this communication.

# 6.0 Implement Asset Base

The *Define Asset Base Architecture* phase of the DAGAR process has produced a formal architecture specification along with the specification of a set of major modules (called realms) into which the major areas of functionality to be provided by the asset base have been partitioned. The ASSET BASE ARCHITECTURE lays out what the key architecture variants are — these are provided by the components that implement the functionality defined by each realm — and depending on how these components are combined, various system architecture possibilities can emerge from the asset base architecture. The realm descriptions define the services and objects that are encapsulated within the realms themselves.

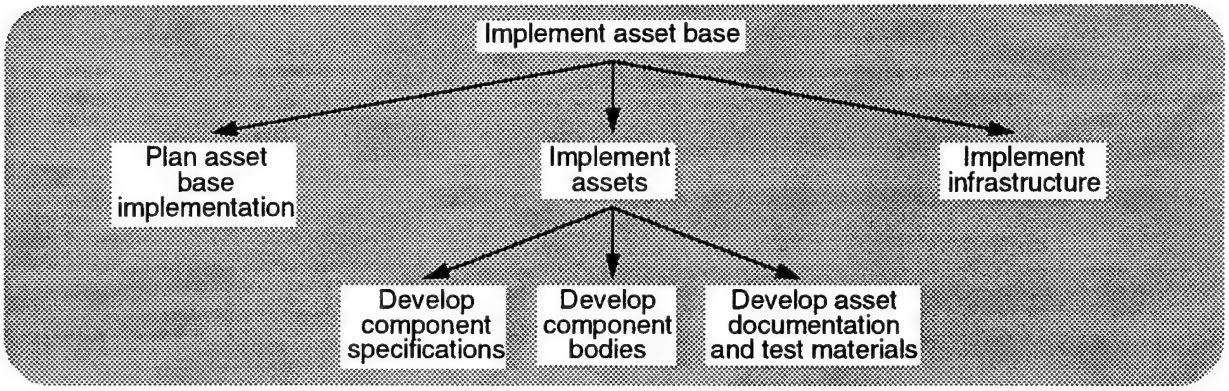



**Exhibit 26.** Implement Asset Base Process Tree

The primary purpose of the *Implement Asset Base* phase of DAGAR is to define and complete implementations of the DAGAR components which exist within each realm. In DAGAR, such component implementations include code fragments for substitution points defined in the realm declarations and complete code, using an extended Ada syntax, which specifies how the services encapsulated in the realm are accomplished. Besides the components themselves, this phase also produces supporting materials such as test plans, documentation templates, and architecture usage rules that help an application engineer develop subsystems in terms of the architecture. These rules are used by the EDGE/Ada toolset in providing system composition guidance to application engineers.

A key challenge in completing this phase will be to manage the tension between architecture discoveries that are made as a result of working on the implementations and a desire to immediately adjust the architecture based on those discoveries. There indeed will be a strong feedback current between *Define Asset Base Architecture* and *Implement Asset Base* but the activities in each of these phases must not be allowed to interact arbitrarily.

## Approach

Once the architectural framework is in place, the detailed work of planning and implementing the assets and infrastructure that collectively make up the asset base can begin. Asset base implementation includes planning the asset base implementation, implementing assets, and implementing the infrastructure that will assist application engineers in selecting assets.

Exhibit 27, gives an overview of the *Implement Asset Base* phase. While the primary activity of this phase is implementation, it should not be understood as the simple production of code such as takes place during the implementation of a single application in Ada. Rather, implementation must take into account a potential multitude of application contexts within which assets available

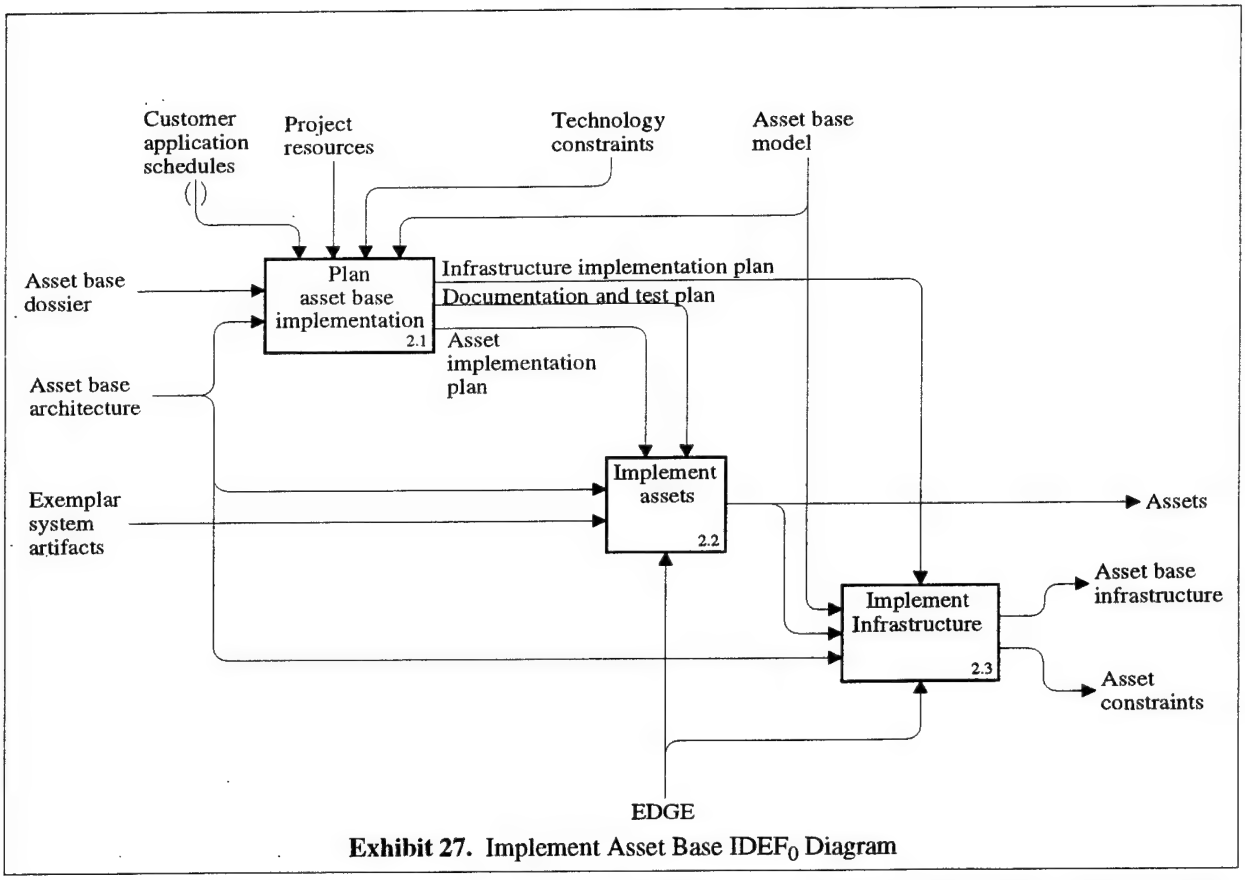



**Exhibit 27.** Implement Asset Base $IDEF_0$ Diagram

through the asset base can be applied. But these contexts of use cannot be arbitrary; they must fit within the scope of the asset base architecture.

The DAGAR approach is based on the following two tenets:

- An asset base, devoid of its architecture, is not intended to be generally useful and reusable.
- Asset base success cannot be measured apart from the architecture within which it was created.

These ideas may be rather startling since some reuse literature gives the impression that reuse of components (assets) in arbitrary contexts is what is important. DAGAR asserts that assets are constructed to be useful in a specific domain and architecture context. They are not considered to be reusable outside this context, although an asset base user would not be prevented from extending the area of applicability. The domain engineer is concerned only with supporting explicit contexts and has nothing to say outside of these contexts. DAGAR-produced source code is generated from higher-level specifications (realms and components) that are part of an architectural framework. Generation of compilable and linkable source code takes place using properties and relationships that are recorded within the framework. The user must make certain choices in order to produce products that are ready for use within a particular system. With a sufficiently rich and variable architecture, there are many different system contexts within which the assets available through the architecture can be used.

## Results

Another important aspect of the DAGAR approach to implementation is that the process includes more than completion of component specifications and bodies. DAGAR requires that up front effort be applied in developing both assets, supporting materials such as documentation and test plans, along with component composition rules. The payoff for this extra effort comes from being able to automate generation of subsystem after subsystem from the domain asset base. While there are costs for establishing an asset base, there will be significant application cost reductions for subsequent systems using the domain. With the infrastructure provided by EDGE/Ada for selecting components, new subsystems can be created with relatively little effort and therefore with great cost reductions compared to producing the subsystems from scratch, using traditional application engineering methods. The Army STARS Project Experience Report [16] identifies significant cost saving opportunities that can be realized using the DAGAR approach. Each new application of the domain will increase the reliability and stability of the asset base as assets are reused and repeatedly tested in a variety of circumstances.

## Process

As shown in Exhibit 27, the *Implement Asset Base* phase consists of three main sub-processes:

- In *Plan Asset Base Implementation,* domain engineers consider the best way to complete the implementation of the components identified as being part of the architecture. These considerations include which of the components to implement first, whether it is better to take a breadth-first vs. depth-first approach to the implementation of components, and what are the minimal supporting materials (test plans, test cases, documentation templates, etc.) required for each component. Plans must also include how and when to complete the component composition rules that help application engineers to decide which components are to be extracted from the asset base.

- The primary goal of the *Implement Assets* sub-phase is to complete the extended Ada definitions of the components that form the core of the Asset Base itself. These definitions include component-specific tailoring of the realm specifications produced during the *Define Asset Base Architecture* phase (these are the EDGE/Ada component specifications) and the specification of the details of how the component provides the services required of the realm to which the component belongs. Components can use the services available in other realms if they are declared to have realm parameters within the ARCHITECTURE SPECIFICATION. This sub-phase is also the site where asset base supportive materials such as test plans and cases are developed.

- In the *Implement Infrastructure* task, domain engineers produce extensions (if any) to the EDGE/Ada toolset along with data to be made available to application engineers as they interact with the toolset to obtain assets for use as part of the application being built. Primarily, this infrastructure data takes the form of component composition rules that prevent any inappropriate component interactions and/or connections from occurring during application development.

## Sequencing

- Basic Sequence. In most cases, the *Plan Asset Base Implementation* sub-phase should be completed before the other two sub-phases begin. In particular, any phasing strategy covering the sequencing and completeness of component development should be understood before asset implementation gets underway in earnest.

- Co-development of assets and infrastructure. Since many component composition rules involve either mandatory or prohibited relationships among particular components or groups of components, it may be preferable to address the creation and precise formulation of these rules in concert with the development of the components themselves. If there is a delay between asset and infrastructure development, domain engineers should be careful to record any insights or concerns regarding component interaction for later use during the *Implement Infrastructure* task.

- Toolset-driven asset development. The middle sub-process in the *Implement Asset Base* phase is directly supported by the EDGE/Ada toolset while the other two sub-processes are not directly supported. As such, there may be a tendency to let the tool support overly control the major activities of this phase of DAGAR. There must be sufficient attention paid to all three major sub-processes and the planning results must be respected as asset base core development is carried out during this phase.

## 6.1 Plan Asset Base Implementation

Planning for asset base implementation involves selecting the technology to apply in the development of each asset, planning the implementation of assets, planning for documentation, and planning the test and validation strategy for the assets. Implementation planning includes determining the optimum strategy for incrementally implementing the assets and for maximizing the use of legacy artifacts as much as possible. Factors to be considered include time criticality for providing the assets to application engineers in need of the domain functionality.

Use of DAGAR constrains the method of asset development inasmuch as DAGAR itself is based on the use of generation technology (in particular, the EDGE/Ada toolset) rather than manual construction of assets. Planning must therefore focus on how to best produce components to support this basic generative approach. While a generation approach might not be suitable for every domain, a significant number of domains can effectively be served with the application of DAGAR and its supportive toolset.

Another consideration to be made during planning is the infrastructure that will be provided for application engineers to allow selection of assets from the asset base in a manner appropriate to their application. The EDGE/Ada toolset provides tools to allow the application engineer to automatically select assets based on the asset base architecture.

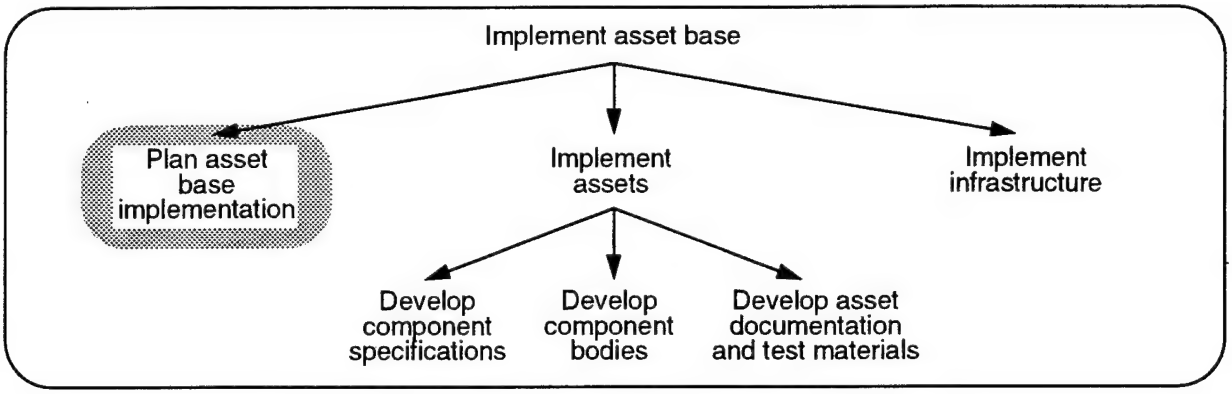



**Exhibit 28.** Plan Asset Base Implementation Process Tree

The primary purpose of the *Plan Asset Base Implementation* task is to consider the ASSET BASE ARCHITECTURE from both development and usage perspectives to identify and arrange a set of component development activities that will produce both assets and supportive materials in a time- and cost-effective manner. The plan must consider both who will be using the assets and when the assets will be needed. For a complex asset base, with many realms and components, it will likely be the case that the completion of asset base products can be staged in a way that accommodates domain engineering resources and asset base customer needs. The various planning documents produced in this task are targeted to match development resources with identified and anticipated asset base customer needs.

The use of DAGAR in one sense contradicts late binding of asset implementation technology choices to individual assets. By giving up late binding time, added support for application engineering in terms of the asset base can be provided. Domain engineers require sufficient training in the methods of generator-based engineering in order to be successful. By grounding the syntax and semantics of the component specification language in Ada, it is expected that engineers with Ada development backgrounds will find transition from Ada to the use of DAGAR to be natural and productive.

## Approach

Although DAGAR presupposes a technology for component implementation, there are still significant choices to be made regarding the sequence and pattern of component development activities. These choices are facilitated by the following suggestions:

- Prioritize component development. In the case of several distinct application engineering customers, decisions regarding which of these customers should have highest priority access to components, and when this access is required, are necessary.
- Consider legacy system artifacts. Where significant legacy system artifacts exist that conform to the essential architectural principles identified during the *Define Asset Base Architecture* phase, it may be possible to reuse adapted versions of these artifacts during the *Implement Asset Base* phase. The planning task should identify these artifacts and consider how and when they can be adapted.
- Incrementally build advanced components from simpler ones. Where several components in the same realm can be designed as successively refined or enriched versions of each other, the simplest component should be created first, with subsequent versions created after the simpler one has been completed and preferably tested. These simpler components should be retained as viable architectural entities even when no currently identified customer exists for such a component. They are useful for testing other parts of the architecture.
- Develop assets and infrastructure in parallel. While it may not always be the case that asset base utilization infrastructure (e.g. component composition rules) can be developed in parallel with the assets themselves, it is still a good idea to plan for this kind of coincidental development. Component interaction constraints will be clearest while the components are under development and these constraints should be formalized as close as possible to their determination.

## Workproducts

■ ASSET IMPLEMENTATION PLAN

This plan contains a development and delivery schedule for each component in each realm in the current ASSET BASE ARCHITECTURE. As much as possible, this plan should carefully and precisely identify when-needed-by and when-finished-by dates for these components as well as resource estimates (e.g. person-hours) required for completion of each component. When specific customers have been identified for components, these customers should be identified by name and (when possible) where in the customer's application development life-cycle (e.g. requirements, design, code, and test) a subsystem made up of components within the architecture is needed. An integration strategy for incorporating the subsystem in the larger application should be included if the application's utilization requirements are sufficiently clear in advance. All resource estimates should take into account time and effort needed to document and test the components in their architectural context, but this plan can defer details to the other plans produced during this task.

■ DOCUMENTATION AND TEST PLAN

This plan supplements the ASSET IMPLEMENTATION PLAN and describes testing and documentation methods and techniques to be followed during *Implement Asset Base*. The level of documentation and testing to be provided should be included in this plan as well as the available resources (e.g. from legacy artifacts and from the *Define Asset Base Architecture* phase) that can be used for testing and documentation purposes during component implementation.

■ INFRASTRUCTURE IMPLEMENTATION PLAN

Asset Base Infrastructure in DAGAR is primarily identified with the EDGE/Ada toolset and the support that is provided by this toolset for domain and application engineering. Any known modifications or extensions to this toolset that have been identified during architecture definition or planning for this sub-phase should be given in the INFRASTRUCTURE IMPLEMENTATION PLAN. It will normally be the case that few, if any, modifications will be required in the toolset. However, the production of specific data structures that are used by the toolset should be detailed in this plan. In particular, component composition rules that limit how components can be put together in terms of the architecture are required and the development of these infrastructure elements should be planned for and scheduled in time to support expected asset base customers.

## When to Start

- ARCHITECTURE SPECIFICATION and REALM SPECIFICATIONS completed. In order for the development of components in DAGAR to commence, their architectural underpinnings must be understood. In DAGAR, these underpinnings are provided by the ARCHITECTURE SPECIFICATION and the various REALM SPECIFICATIONS that are produced during *Define Asset Base Architecture*.

  While a spiral-form of asset base development is possible where some component development begins before the architecture is finalized, there should be enough of the architecture defined so that the component-to-realm interconnectivity supported by DAGAR can be used to establish the necessary associations. For those components with bodies under development, the corresponding realm specification must also be complete. Planning can identify those realms and components that are candidates for spiral-based development.

- Domain Engineering staff with component development skills available. The nature of DAGAR-based asset base construction allows staff with different skill sets and backgrounds to be assigned to the various phases and tasks of asset base implementation. The earlier phases of asset base engineering will typically be completed by staff members with wider backgrounds in both domain knowledge and experience in asset base architectural design. The development of components can take place with less experienced staff as long as they are trained in the extensions to Ada offered by the EDGE/Ada toolset.

- Support materials from prior domain engineering phases identified. It is not necessary to wait until the *Implement Asset Base* phase to begin the work on supporting materials for the assets. The basis for some of these support materials may already have been produced during *Define Asset Base Architecture*. All such materials that can be used as source material for the assets to be developed during this phase should be gathered prior to the start of *Plan Asset Base Implementation*.

## Inputs

- ASSET BASE DOSSIER. Contains usability and feasibility data, as well as traceability to possible prototype artifacts for reengineering.

- ASSET BASE ARCHITECTURE. Overall architecture of the asset base. In DAGAR terms, this consists of the formal ARCHITECTURE SPECIFICATION using a language for this purpose supported by EDGE/Ada and a set of REALM SPECIFICATIONS that describe the services and entities that define each major module that is part of the architecture.

### Controls

- CUSTOMER APPLICATION SCHEDULES. Used to determine the staging strategy for the asset base
- PROJECT RESOURCES. Needed to develop reasonable objectives and schedules for implementing assets.
- TECHNOLOGY CONSTRAINTS. Any global constraints (mandates, limitations) on technology that will affect how the assets and asset base architecture will be used and therefore can impact their development.
- ASSET BASE MODEL. Used to bound the features and customers and interface constraints considered.

### Activities

None of the activities listed in this subsection require that they be performed in a particular order. To the extent that the development of the plans depends on knowledge of customers and customer needs, the first three activities can be profitably arranged to precede the others. The DAGAR asset base implementation planning activity is considerably simplified from the form it takes in ODM because of the pre-selection of a generative approach to asset base engineering. As such, there is no need to plan for the selection of implementation technology.

➤ Consider customer constraints on component usage and assembly into subsystems

While the implementation technology for the asset base itself has been chosen in advance, the inclusion of products obtained from the asset base into a customer's own application context must be considered during asset base planning. The layered architecture as conceived during *Architect Asset Base* reflects a broad base of customers and potential customers. This architecture must now be considered from the perspective of specific customers and detailed customer needs. It may be the case that specific layers or stacks of layers have become increasingly important to a customer and therefore separate configurability of assets from these layers is now required. Specific integration concerns of customers who intend to use the assets in one or more applications should also be listed and addressed. Knowing these concerns can help in planning the phased component development process that will be followed in *Implement Asset Base*.

➤ Update ASSET BASE ARCHITECTURE and REALM SPECIFICATIONS

As a result of the previous activity, changes may have been identified to both of these workproducts. These changes should now be reflected in updated versions of the work products. It may be useful to save both versions of the workproducts as legacy materials from the domain engineering process as a whole to record the observed distance from the architecture as initially proposed and then the architecture as implemented.

➤ Consider customer context constraints on schedule

Consider the current life cycle phase for application projects that are potential utilizers of the assets to be developed. The overall plan should reflect opportunities and conflicts stemming from the schedule of each utilizer project. Adjust the schedule or phasing of implementation if necessary to coincide with these phases. This can make a difference between assets being used or being passed by, whatever their technical merits.

Record the results in the ASSET IMPLEMENTATION PLAN. Place the anticipated domain engineering project schedule with milestones for completed assets on a common time line with the anticipated schedules of potential customer applications. A separate phasing strategy may be required for each existing or anticipated system into which domain assets are to be incorporated.

> Example. One project might be in an early requirements definition and negotiation phase, where initial versions of domain assets could be used in a prototyping capacity. Another project may be almost through with detailed design, and therefore only be able to make use of thoroughly tested components that offer a close match to the functional profiles assumed by the current application architecture. Each life cycle entry point presents different challenges.

➤ Plan development stages

Based on the schedule and feasibility estimates for the components within each of the realms, develop the detailed staging strategy for implementing the components. Consider factors such as requirements/opportunities for migrating components and subsystems of components into existing systems; use of existing systems as testbeds/validation suites, and advantages of early release of certain configured components to facilitate adoption by particular customers. On a per-component basis, the staging strategies can include any of the following options:

- *Prototype components.* Prototype components are those meant to validate the both the DOMAIN MODEL on which the architecture is based as well as the architecture and candidate inter-realm dependencies within the architecture. These early components may not be carried forward into the final ASSET BASE. For a realm with a large number of components, a cross-section of the these components may be scheduled for simultaneous development to test the usability of the variability supported by the components within the realm.

- *Component assemblies to validate utilization.* A vertical slice through the architecture in the form of components adequate for incorporation into a complete subsystem is highly desirable before attempting full-scale implementation of all (or most) components in the asset base. The key is that enough of the components need to be implemented that application developers can experience the direct impact of having sufficiently many components for use within the application.

- *Components to validate infrastructure.* A set of components may be selected for development that, taken together, will allow for validation of key aspects of component-to-component dependencies that will be encoded and applied by the EDGE/Ada toolset to support application development. As additional components are developed that allow for more application-developer choices as presented through the toolset, effective use of the toolset, and the adequacy of rules governing component selection, can be verified.

➤ Plan asset base documentation, testing and validation

Depending to some extent on the staged development plan, plan the strategy for documenting, testing and validating both individual components and the asset base as a whole. Document this in the DOCUMENTATION AND TEST PLAN. The following paragraphs highlight some points to consider.

- *Testing components is different that testing source code modules.* As mentioned above, the syntax of the language supported by EDGE/Ada to specify components is largely that of Ada. However, since components are written with explicit use of other realms in the DAGAR domain architecture, and in turn are meant to be used by components in other realms, their testing and documentation should primarily address their role and function in combination with other components that serve as clients of, and resources for, the current component.

Taken to one extreme, stand-alone testing of components can be viewed as having minimal value. Some components are written to depend on no other realms in the architecture and such components should be tested in comparable isolation from the architecture. But even here, it is better that test harness components be developed and included as part of the architecture itself so that testing happens within the scope of the architecture. Test plans and data sets can also be developed in relative independence of the architecture, but they should be applied and evaluated from an architectural perspective.

- *Base component documentation and test materials on architecture materials.* Components in one sense are instances of the realms to which they belong. During architecture definition and in particular during the *Develop Architecture Documentation and Test Materials* task, sufficient information is available that can be used to formulate test plans, policies, procedures and even test data. Moreover, documentation templates for each realm can be established that need to be refined for each component within the realm. These base materials should be the first source of information to be consulted in connection with this planning activity. Inasmuch as the ASSET BASE ARCHITECTURE is based on the DOMAIN MODEL, supporting test and documentation material should also be based on the DOMAIN MODEL. In planning the refinement of test and documentation material for particular components, it is advisable that the architectural relationships be consulted so that testing materials can be planned that make use of these relationships. Such test materials will naturally exhibit a high degree of reuse and will feature test case data designed to show the essential variability captured in the individual components.

- *Some supporting materials should be developed as assets themselves.* The application engineer will finally need to test the extracted components and/or subsystems in the target context (e.g. hardware, system software, other application software) in which the components or subsystem will be used. Documentation to support this use will also be required. The asset base should be structured so that this material is available using the same (or an integrated) mechanism used to extract the components themselves. When viewed from this perspective, these materials should be considered as assets themselves. Planning should take into consideration how to prepare such materials for inclusion in the asset base.

➤ Plan asset base infrastructure development

The asset base infrastructure will, for the most part, be restricted to the component composition rules that are used when an application engineer is using the EDGE/Ada toolset to identify assets for use within the application. Determining when these rules will need to be developed, and in what customer contexts they are expected to play a role, is the primary function of this activity. The rules themselves will be determined as the construction of the components proceeds and fine-grained component-to-component interactions and dependencies are identified during component implementation. Clearly, the rules along with any changes to the EDGE/Ada toolset to support the current domain engineering project and sets of application engineering projects will need to be completed by the time these projects require the functionality provided by the respective changes and additions.

➤ Plan asset evolution

In general, the production of assets and modifications to the Asset Base Architecture will not end once the domain engineering effort that first establishes the architecture has finished its work. Plans should be established that indicate how and when results produced from application engineering activities can be fed back into the asset base. Most domain engineering projects will establish a set of maintenance and asset base management procedures and will secure resources that will enact these procedures. This activity establishes how feedback from asset base application will flow into these procedures.

### When to Stop

- All three planning documents completed. Three particular planning documents have been identified as outputs for this task. Sufficiently clear and informative versions of these documents are required before the *Plan Asset Base Implementation* task can be stopped.
- A schedule for implementing assets developed. All good plans require a schedule by which the activities that comprise the effort being planned can be clocked and monitored. In regard to the *Implement Asset Base* phase, the schedule must be sufficiently detailed and precise that all identified customer application projects are assured of having their needs met. The schedule must also consider project resource constraints as well as the estimated feasibility of the components themselves.

### Guidelines

- Consider component interconnection trade-offs. The simplest components are the ones that are self-contained; they have no dependencies on other realms in the architecture. It may seem that the best place to start component implementation is with these components. However, because the DAGAR process crucially depends on components working together in an architectural context, it may be best to schedule component implementation so that certain of these connections are used and tested early. It would be best not to pick components with many connections to other realms in the architecture. But, as long as customer need for components can be accommodated, completing components in three or more layers of the architecture that contain references to one another will be valuable in evaluating the architecture and the ability of the domain engineering team to work correctly using DAGAR principles and tools.
- Consider component evolution strategies. In many cases, realms will contain components that are related to one another in terms of features or performance. Components will differ in that they will be faster or rely on different resources (e.g. identified by the realm parameters named for the component) in completing the services identified in the Realm Specification. Some components may be defined to depend on the realm that they are in (these are the so-called symmetric components). In these cases, it is advisable to work on the simpler forms of the components first and later move on to the more complex ones. The more complex cases may turn out to use the same (or nearly the same) code as the simpler component for some of the defined services. It may even make sense to provide component body stubs for some of the realm's services and get a single subsystem involving the component working with minimal functionality. This minimal form of the component can then be used in component integration testing that will validate the asset base infrastructure.
- Consider adding scaffolding realms to architecture. It may be necessary to propose additional realms for inclusion within the architecture to serve as intermediate testing platforms for components (or several component layers) that otherwise cannot be tested until a complete subsystem integration is performed. As an architecture may contain many layers, postponing testing in this way will impact quality assurance on the lower levels of the architecture. Even though there may not be any application engineering customers for these scaffolding realms, it is advisable to permanently add them to the architecture so that as other components are added, they can be tested using the same test components within these realms.
- Consider test plans and other supporting materials as assets. The addition of test and document work products to the asset base itself was proposed earlier in this section. While it may not make sense for these materials to be treated as stand-alone assets, apart from the component(s) that require or suggested them, they should at least be viewed as bound additions to the components to be provided to asset base customers who select the components for use in

their application. While the testing materials may originate in templates and drafts first developed as part of the architecture, component implementation details will cause significant extensions to these base materials. These modifications will be reflected in the supporting materials that are to be treated as collateral assets for inclusion in the asset base. Analogously to components being architecturally related to one another, test and documentation materials can have similar relationships to one another. These relationships should be reflected in the asset base infrastructure used to select and extract assets from the asset base.

## 6.2 Implement Assets

After developing the asset base architecture using the facilities of EDGE/Ada, and then producing a plan for how to complete development of the asset base, the domain engineer is ready to begin the task that in some sense is the primary goal of the entire domain engineering activity; namely, the production of the assets themselves. EDGE/Ada provides both a specification language and tool support to be used in component development. The specification language is an extension of Ada so that engineers who are familiar with Ada can be quickly trained to develop components. REALM SPECIFICATIONS developed during the *Define Asset Base Architecture* phase provide the starting point for component development. Support is also provided within EDGE/Ada for developing asset utilization infrastructure that help application developers understand and apply the assets contained in the asset base.

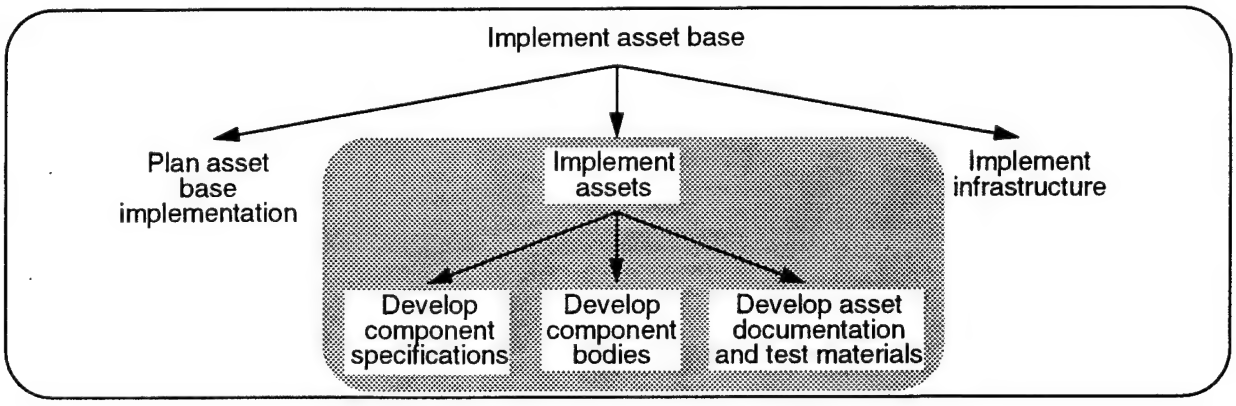



**Exhibit 29.** Implement Assets Process Tree

The primary purpose of the *Implement Assets* sub-phase of DAGAR is to produce a set of component specifications and component bodies that conform to and implement the ASSET BASE ARCHITECTURE. These components are designed to provide the desired variability within the architecture for required and anticipated features and capabilities in the context of a set of asset base customers. These features and the necessary understanding of asset base customer wants and needs comes through domain analysis and modeling.

Although the specification language syntax is very close to Ada, the task of producing component specifications and bodies is not exactly like that of producing Ada package specifications and bodies. In fact, the more experienced engineers are in producing Ada package code, the more difficulty they may find in perceiving the larger picture that underlies the DAGAR approach. Producing quality components requires that this picture be understood and applied during the development process. So, while Ada experience is desirable, the component development process will need to be monitored to guard against an overly Ada and system-specific mindset while the *Implement Assets* phase is underway.

### Approach

Component definitions produced during this sub-phase are implementations of the architecture modules (realms) produced during *Define Asset Base Architecture*. ***Components*** are thus instances of the realms that were developed as part of the ASSET BASE ARCHITECTURE and consist of a specification part and a body part.

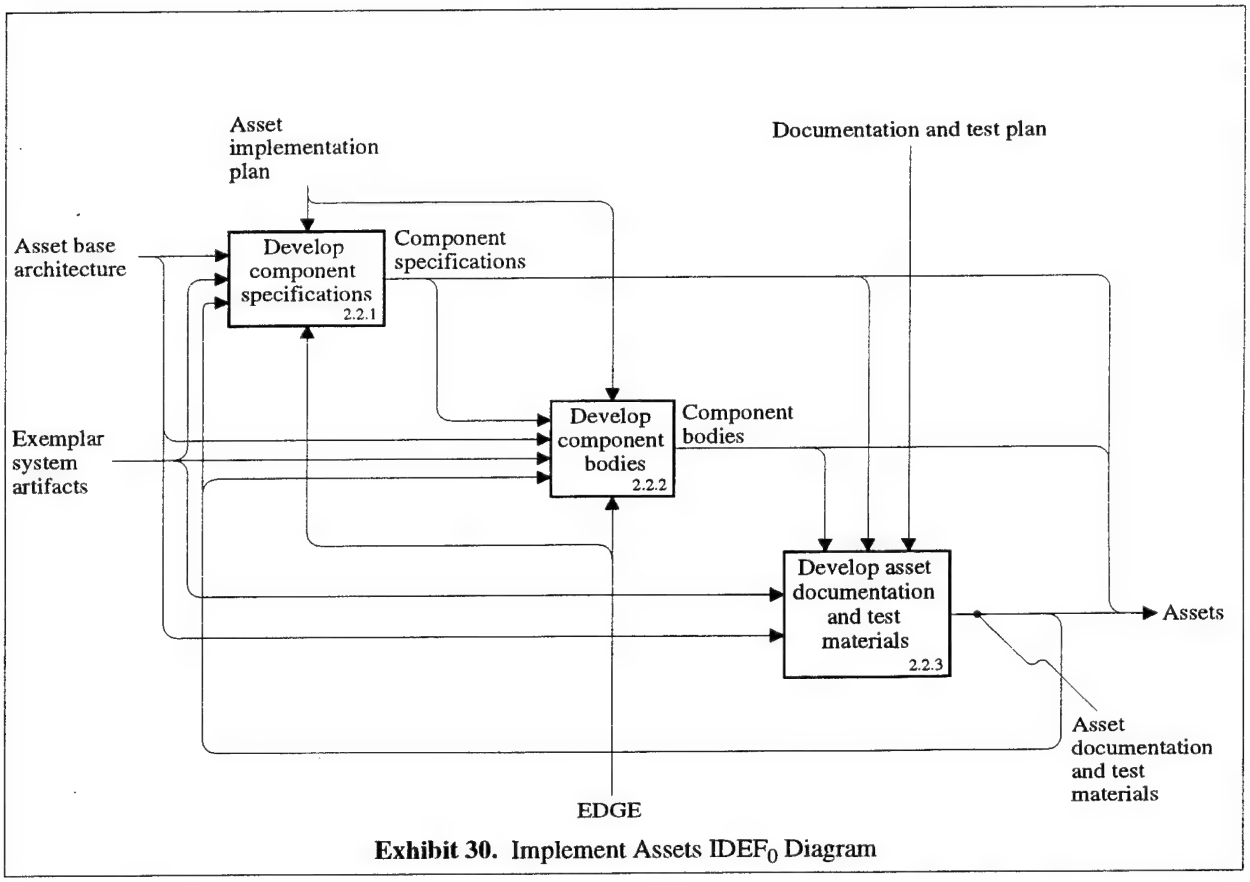



**Exhibit 30.** Implement Assets $IDEF_0$ Diagram

A ***Component Specification*** consists of fragments of extended Ada code that provide implementation details that supplement the realm interface defined in the REALM SPECIFICATION. Because the definition of a component can make use of the services provided by any of the realm parameters declared for the component, these fragments can be used to introduce dependencies at the component specification level that are satisfied in terms of these realms.

A ***Component Body*** consists of relatively complete sections of extended Ada code that define how a component implements the services and other interface elements of the realm to which it belongs. EDGE/Ada syntax provides for the symbolic reference to the services and interface elements for any of the component's realm parameters. These references are called ***holes***. At code generation time (as required by the application engineer or the domain engineer for testing purposes), these holes are filled by references to components that instantiate the realm parameters. These instantiations provide the means to generate complete Ada code that binds together definitions at both the realm and component levels.

Exhibit 30 provides an overview of the *Implement Assets* sub-phase. The tasks within the process diagram will be covered in subsequent subsections. It is important to note that toolset support is provided for processing the component specifications and bodies and producing compilable Ada from them. At the current time, no direct support is provided in the toolset for producing the supporting materials such as component documentation and test materials.

It is assumed that the domain engineering team is sufficiently knowledgeable about Ada to be able to complete the DAGAR component bodies. As noted previously in this guidebook, DAGAR is implemented in Ada and is targeted to the production of Ada products. To be successful, the domain engineering team must either have sufficient Ada knowledge or be trained in Ada prior to the start of the DAGAR development process. This Ada knowledge must be tempered with

awareness of the differences inherent in DAGAR component development compared to standard package design and implementation in Ada.

## Results

Whereas, the principal product of the *Define Asset base Architecture phase* was a specification of *what* functionality is to be provided by the asset base, the *Implement Assets* sub-phase produces the implementation describing *how* this functionality is accomplished. Along with the implementations themselves, supporting materials necessary to verify the components and to describe how they work internally and with respect to one another are also produced. These results provide the core, end-user product to be produced by a domain engineering effort.

## Process

As shown in Exhibit 30, the *Implement Assets* sub-phase consists of three main subtasks:

- In *Develop Component Specifications,* domain engineers elaborate the REALM SPECIFICATIONS that establish the core services and capabilities of the asset base with COMPONENT SPECIFICATIONS that provide details to specialize the realm definition and reflect the various alternative implementation and connectivity choices available in the architecture.
- The primary goals of the second task, *Develop Component Bodies*, is to define with sufficient detail how the services published by each realm are actually carried out. The language used for Component Bodies is basically Ada with extensions to support the use and instantiation of holes where services and definitions from realms that parameterize the component can be included in the component body.
- In the third task, *Develop Asset Documentation and Test Materials*, domain engineers produce the supporting material that is necessary to understand and evaluate the component being produced. In addition to supporting a single component, material that describes how components work together to achieve certain goals, and how to test that these goals are achieved, is also necessary.

The EDGE/Ada toolset is targeted to provide effective engineering support in executing all three of these tasks, although there is stronger support in the current toolset for the first two of the subtasks

## Sequencing

There is no absolute requirement that the task elements of Implement Assets be performed in a particular order. While a pattern of developing each component specification and body jointly can work, under some circumstances it may be preferable to work on the specification level for a large number of components before proceeding to any significant body development.

- Supporting material with component definitions. In order that knowledge gained during development not be lost, it is advisable that there not be a long delay between the time component specifications and bodies are produced, and the time supporting materials are developed. Even if different personnel (e.g. documentation or quality assurance specialists) are assigned to the task of producing supporting materials, the time at which these materials is produced should coincide, as far as possible, with the time at which the specifications and bodies are completed. Not doing so can lead to valuable information gained during development being lost when it comes time to verify and document what was developed.

- Component development sequence according to plan. The planning phase for *Implement Asset Base* can be expected to drive the sequence of component development to a large degree. For those components for which an early working version is required, it will be necessary to follow a component specification with a rapidly produced body for that component. The ASSET IMPLEMENTATION PLAN should be considered the primary input in deciding component development steps and sequencing.

## 6.2.1 Develop Component Specifications

Although it is not necessary for either of the phases *Develop Asset Base Architecture* or *Plan Asset Base Implementation* to be complete, there must be a sufficient architectural basis and a plan of attack for how to sequence component development before the *Develop Component Specifications* task can begin. But, due to the nature of DAGAR, the work to be done to complete a COMPONENT SPECIFICATION is relatively simple and quick. The EDGE/Ada capabilities make it easy to verify if a component specification is consistent with its realm.

By binding each of the realm parameters for a component with a valid component specification within the realm, EDGE/Ada is able to generate compilable Ada code that implements an architectural binding for the component. Doing an Ada compilation will then indicate whether there are any syntactic errors in either the Realm Specification or the Component Specification. Errors in either location can then be corrected.

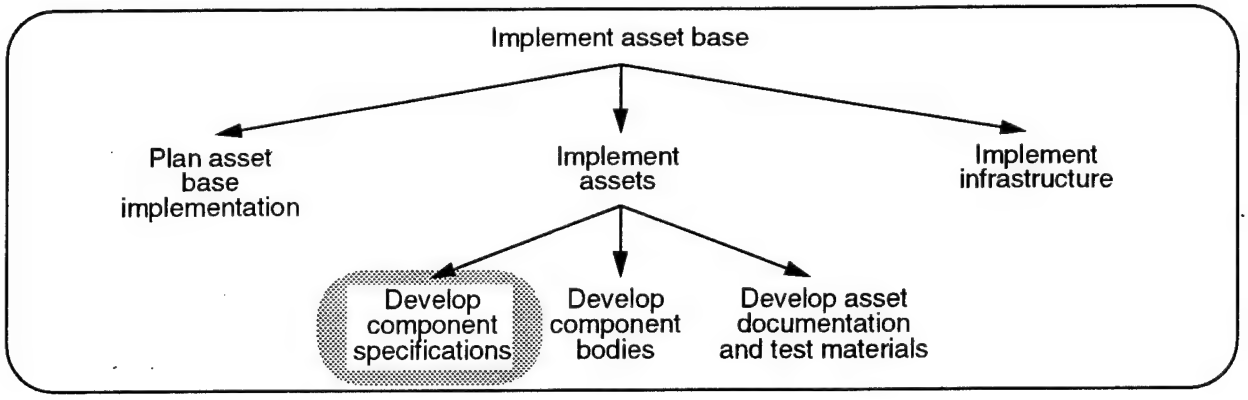



**Exhibit 31.** Develop Component Specifications Process Tree

The primary purpose of the *Develop Component Specifications* task is to produce a sufficient number of COMPONENT SPECIFICATIONS, written as extended Ada fragments, to allow further work on Asset Base development to proceed. A spiral development pattern is possible where a small number of component specifications are completed, followed by the necessary component bodies, and followed by an integration of related components to verify both compositional correctness as well as functional correctness for the subsystem.

A significant challenge to be addressed in this task is that COMPONENT SPECIFICATIONS, by themselves, are relatively useless as they are meant to add detail to existing REALM SPECIFICATIONS. As such, they cannot be tested or evaluated apart from the realm specification that they extend. This concern mirrors one discussed in the *Develop Realm Descriptions* task that indicated that realms cannot be tested in stand-alone fashion as they require components to provide fillers for the holes left in the realm specification. Engineers responsible for testing at each of the realm and component specification steps of the DAGAR process will need to coordinate their work closely. One possibility would be that an early iteration of the *Develop Component Specifications* task be completed as each realm is completed.

## Approach

The essential distinction between Component Specification and Body was made previously in the opening paragraphs defining *Implement Assets*. Looking at each realm (following an order obtained from the Asset Development Plan), and then looking at selected components for these realms, the domain engineer must write extended Ada code fragments that fill in the missing details within the Realm Specification. These fragments are placed in separate specification files and checked into the EDGE/Ada database. EDGE/Ada verifies the correctness of the file by checking whether all holes that need to be filled in the realm have data supplied by the Component Specification. If a component itself has any realm parameters that need to be matched with components, the EDGE/Ada user must make sure that a parameter/argument binding for each of these parameters has been made. If any of these parameters are used within the component specification, EDGE/Ada will not be able to generate Ada unless a binding has been specified. Depending on the ASSET IMPLEMENTATION PLAN, a domain engineer can elect to complete most, if not all, of the COMPONENT SPECIFICATIONS before proceeding into the implementation of the COMPONENT BODIES; or, the specifications and bodies can be implemented together in an Asset Base evolution spiral.

## Workproducts

- COMPONENT SPECIFICATIONS

As discussed above, a COMPONENT SPECIFICATION declares implementation-specific choices that a component makes regarding how a REALM SPECIFICATION is to be translated into code that is integrable into a subsystem as desired by an Asset Base customer. Exhibit 32 shows an example of the kind of full declarations and declaration fragments that can be placed in a COMPONENT SPECIFICATION. This example is compatible with the REALM SPECIFICATION example shown in Exhibit 24. Each component has a name (indicated by the **$component keyword**) and each full or fragment declaration also has a name (introduced by the **$for** keyword) that corresponds to a named hole inside of the component's REALM SPECIFICATION. A component can leave a hole empty if desired as shown for the hole **Symmetric_Function_Declaration**. Realm parame-

```
with {Vectors};
with {{Computation_Support}}; use {{Computation_Support}};

$component Std_3_Matrices is

$for Vector_Declaration use
   {Vectors}.Vector

$for Matrix_Index_Declaration use

      type Matrix_Index is new Integer range 1..3;

$for Matrix_Declaration use

     type Matrix is array
       (Matrix_Index, Matrix_Index) of Real;

$for Symmetric_Matrix_Declaration use
   {{Computation_Support}}.Symmetric_3_x_3_Matrix

$for Symmetric_Function_Declaration use

$end Std_3_Matrices;
```

**Exhibit 32.** Complete Matrix Component Specification

ters are indicated by a single pair of braces (e.g. **{Vectors}**) while supporting parameters available at the entire architecture level are indicated by pairs of double braces (e.g. **{{Computation_Support}}**). Substitution for any of these parameters is handled by EDGE/Ada.

### When to Start

- REALM SPECIFICATION complete. The most basic entrance requirement for this task is that a REALM SPECIFICATION exist for each component that is under construction. It is not necessary that the entire ASSET BASE ARCHITECTURE be complete. Because components can in turn depend on other realms (which in turn contain other components), it is advisable that an architectural cross-section consisting of the ARCHITECTURE SPECIFICATION and a companion number of REALM SPECIFICATIONS should be in place before any significant COMPONENT SPECIFICATION development begins.

- ASSET IMPLEMENTATION PLAN sufficiently complete. Component implementation should always take place with guidance as contained in the ASSET IMPLEMENTATION PLAN. The plan can itself be undergoing revisions, but enough information needs to be recorded so that current component development will take place in a manner that best supports overall Asset Base goals.

### Inputs

- ASSET BASE ARCHITECTURE. The architecture consists of an ARCHITECTURE SPECIFICATION that defines the realm and component members that make up the architecture framework and for each realm, a set of REALM SPECIFICATIONS that define the entities and services exported by each realm.

- EXEMPLAR SYSTEM ARTIFACTS. Candidate artifacts that suggest component specification elements and which can possibly be adapted for use as parts of COMPONENT SPECIFICATIONS.

- ASSET DOCUMENTATION AND TEST MATERIALS. These materials are those developed during the *Define Asset Base Architecture* phase that provide template sources for the more complete documentation and test materials to be developed during *Implement Assets*.

### Controls

- ASSET IMPLEMENTATION PLAN. As explained above, essential guidance for this task is provided by the ASSET IMPLEMENTATION PLAN including the order in which COMPONENT SPECIFICATIONS are developed and their synchronization with COMPONENT BODY development.

### Activities

➤ Write each component specification.

The *Develop Component Specifications* task is the first one of those making up the *Implement Assets* phase that addresses implementation details of the asset base — *how* things get done. The first kind of how concerns alternative ways of providing details at the interface level of the architecture. Interfaces as given in a REALM SPECIFICATION are complete in substance but allow for variability in terms of details. The developer of a COMPONENT SPECIFICATION can see the alternatives available by looking at the holes left in the REALM SPECIFICATION. In a straight-forward

sense, the component developer's job is to design appropriate "filler" material by which the holes are completely filled in.

> Example. Exhibit 32 shows an example of a complete component specification for a matrix realm component. As shown in the figures, the syntax for components is based on the Ada language. In the example, Ada type declaration text is provided for the basic types that are part of the matrix realm specification. The component specification makes use of its single realm parameter **{Vectors}** in completing these type declarations. The names in double braces are architecture-wide configurability options that contain basic definitions that are independent of particular realms and components.

➤ Test each component specification.

Testing of component specifications and bodies independently of their integration into client applications is also required. One important mechanism that can be used to test the asset base is the creation of a top-level test component within the realm serving as the primary interface for applications requiring asset base services. The COMPONENT SPECIFICATION for this component should be developed early during asset base evolution and kept up-to-date as the implementation of the asset base proceeds. The EDGE/Ada toolset is used to process all components and, when combined with the appropriate realms, turn them into compilable Ada packages and subprograms.

EDGE/Ada is configurable in terms of how it supports the production of Ada from realms and components. At any time, the domain engineer can elect to produce complete Ada from the evolving architecture. This Ada code can be compiled automatically or, if some details necessary for successful compilation remain to be specified, the engineer can defer the compilation. As soon as all realm specification holes have been completely defined in one or more COMPONENT SPECIFICATIONS, use EDGE/Ada is to produce compilable Ada library units that are then compiled and checked for errors. See Figure 44 on page 106 for an example of the Ada code specification produced for **Std_3_Matrices** component specification.

## When to Stop

- Asset Base Specification thread complete. Depending on the domain engineering team's implementation strategy, it may be best to stop the *Develop Component Specifications* task as soon as a sufficient collection of COMPONENT SPECIFICATIONS has been completed to permit the team to move on to the *Develop Component Bodies* task. If an evolutionary spiral implementation strategy is being followed, this does not mean that the *Develop Component Specifications* is over. This task will continue when the next round of component development is ready to begin. For a small asset base, or one in which the component collection is well understood and stable, it may be possible to complete all of the required Component Specifications before beginning work on any of the COMPONENT BODIES.

- ASSET IMPLEMENTATION PLAN goals reached. The plan by which asset base development is proceeding should be the main control on when a development task is finished (at least temporarily) and another task can begin. The attainment of goals defined in the plan indicates when and if moving on to a new task or phase is permissible.

## Guidelines

- Consider evolutionary component development. In the discussion of this task, there have been numerous mentions of allowing for evolutionary development of the asset base. In fact, software development experiences suggest that such an evolutionary path may be preferred. By not spending too much time at high level considerations (e.g., the interface specification

level only) and driving down to the component body implementation level for key components and then integrating a complete suite of realms, component specifications and component bodies into a complete (if not fully functional) subsystem, the asset base architecture will receive a well-timed evaluation. Problems that exist at higher levels in the asset base can be caught early while remedial action is still possible without severe budget or schedule impact. The domain engineering team should be encouraged in the use of this approach by orienting the ASSET IMPLEMENTATION PLAN toward its support.

- Use existing artifacts for guidance. Most asset base development efforts will, at least in part, be targeted to the support of existing systems with the possible "back-filling" of assets into existing system contexts. As such, features and functionality expected in these contexts must be supported in the asset base. By directly considering available artifacts (especially if they are implemented in Ada) during both component specification and body development, it is more likely that easy integration of assets into these contexts will take place. Due to the languages supported by EDGE/Ada, these artifacts cannot be used as-is, but can still serve as useful inspiration and developmental models for the assets that will be constructed during domain engineering.

- Stick to REALM SPECIFICATIONS. During COMPONENT SPECIFICATION development, the focus should be on providing just those code fragments that are needed to tailor the defined realm specifications and permit complete and correct Ada code to be generated from the combination of associated realm and component specifications. At this stage, there is no need or room for innovation and fault-finding with the capabilities being provided at the REALM SPECIFICATION level. During integration testing when realms and components are combined to yield evaluatable subsystems, inadequacies may arise that need to be addressed. But the Develop Component Specifications task is not intended to provide this opportunity.

- Use EDGE/Ada toolset to its fullest extent. The EDGE/Ada toolset provides a powerful set of processing elements that the domain engineer should use as a routine part of the asset development process. Assets from the DAGAR viewpoint are uniquely supported by the toolset and all aspects of their development should be controlled by executing the appropriate member of the toolset and updating the internal asset development database being maintained by the toolset. The toolset enforces consistency among the various asset base elements and identifies errors. Domain engineers should get in the habit of frequently applying the toolset as they work on the various specification files that make up the asset base. Errors will be caught sooner rather than later and problems with higher levels of the asset base architecture can be corrected while there are fewer repercussions to completed, but as yet unprocessed, lower levels.

## 6.2.2 Develop Component Bodies

The majority of the work in the *Implement Assets* phase is likely to be found here in *Develop Component Bodies*. After a sufficient number of COMPONENT SPECIFICATIONS have been completed to flesh out the REALM SPECIFICATIONS for the realms to which the components belong, the detailed implementation steps to carry out the operations defined for each realm must be written. Component Bodies contain these detailed instructions. They are written in an Extended Ada language supported by the EDGE/Ada toolset. However, unlike the code fragments that make up Component Specifications, Component Bodies are relatively complete descriptions of algorithmic processing that are comparable to Ada package bodies that, due to the nature of EDGE/Ada, they closely resemble. It is up to the ASSET IMPLEMENTATION PLAN to specify whether the work of defining these bodies follows a relatively complete *Develop Component Specifications* task or is integrated with a planned sequence of component development where COMPONENT SPECIFICATIONS and BODIES are completed together.

A component body is not meant to stand-alone in the asset base. Its meaning is found in combination with a REALM SPECIFICATION and a corresponding COMPONENT SPECIFICATION. For those components that are declared to have realm parameters, any interface element from each of these realms can be used to accomplish what is required of the component. While a COMPONENT SPECIFICATION need not make use of any of its realm parameters, there must be some place in the COMPONENT BODY that at least one of the realm's exported entities is required to complete the job allocated to the component — otherwise, there would have been no reason to include the realm as a parameter.

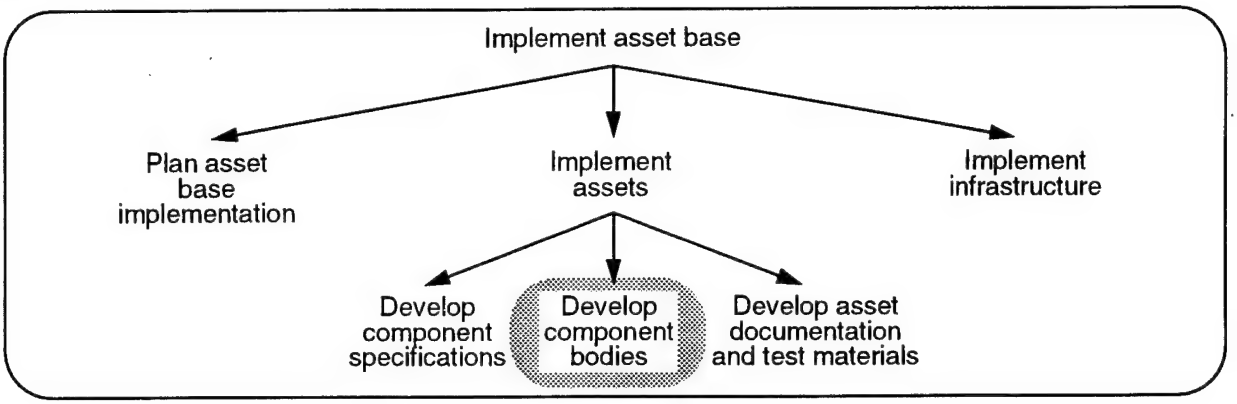



**Exhibit 33.** Develop Component Bodies Process Tree

The primary purpose of the *Develop Component Bodies* task is to complete a COMPONENT BODY for every one of the components identified as being part of the ASSET BASE ARCHITECTURE. In some sense, these body elements provide the core capabilities of the Asset Base itself. For an asset base that is intended to be used to produce code that is then integrated into a larger application, without these bodies, there would be nothing to integrate. This task is the one that is most like typical Ada code development in that the areas of concern are familiar to people trained as Ada programmers.

This familiarity is also a potential trouble spot. If domain engineers adopt too much of an Ada mindset, they may be unable to view their job with a sufficiently broad point of view so that components can be written to be independent of other implementations; i.e., other components, or at least assumed implications based on these implementations. The extended Ada dialect supported by EDGE/Ada permits the engineer to access realm-supplied services from other realms. But the implementations of these realms can be highly variable and during testing, when realm parameters are instantiated by actual components, it may be possible to infer that behavior that is caused by the use of a particular component may be in effect for all components. The only way to be sure about such behavior is to carefully test for all components that may be used as substitutions for the realm parameter. This can significantly complicate testing when several components, each of which has several realm parameters, are integrated together as subsystems that conform to the architecture.

## Approach

Once the extended Ada dialect is familiar to the component developers, the Realm Specification and the Component Specification for the component under construction are examined to learn what is required of the component implementation. For a component that in turn depends on other realms, it will be necessary to consider the impact of these dependencies and arrange for the development of at least one component in each of these realms so that code generation for the current component will be possible. This code generation need not take place immediately but is nec-

```
------------------------------------------------------------------------
---
-- This body implements the matrix operations for any 3*3 matrix
------------------------------------------------------------------------
---

$component body Std_3_Matrices is

------------------------------------------------------------------------
---
-- ...
------------------------------------------------------------------------
---
   function "*" (Left  : in Matrix;
                 Right : in {Vectors}.Vector) return {Vectors}.Vector is

      Vector_Index : Cartesian_Coordinates :=
                           Cartesian_Coordinates'First;
      Result       : {Vectors}.Vector;

   begin -- "*"

      for Column_Index in Matrix_Index loop
         Result (Cartesian_Coordinates'Val (Column_Index - 1)) := 0.0;
         for Row_Index in Matrix_Index loop
            Result (Cartesian_Coordinates'Val (Column_Index - 1)) :=

               Result (Cartesian_Coordinates'Val (Column_Index - 1)) +
               (
                Left   (Column_Index, Row_Index) *
                Right  (Cartesian_Coordinates'Val (Row_Index - 1))
                );
         end loop;
      end loop;
```

**Exhibit 34.** Partial Matrix Component Body

essary for integration and functional testing. EGDE/Ada will be used to process each COMPONENT BODY and verify its compatibility with the current architecture. EDGE/Ada will also be used to complete code generation when specific components are assigned to fill the realm parameter slots for the component under construction. The combination of a REALM SPECIFICATION, COMPONENT SPECIFICATION, and COMPONENT BODY for the component under construction, along with realm and component specifications for each of the realm parameters required by the component, are necessary to enable completion of code generation.

## Workproducts

■ COMPONENT BODIES

COMPONENT BODIES complete the concretization of how a realm is implemented that was begun with the production of the corresponding COMPONENT SPECIFICATION. Exhibit 34 contains a portion of a COMPONENT BODY that corresponds to the COMPONENT SPECIFICATION shown in Exhibit 32. The name of the component is required (here **`Std_3_Matrices`**) and must match that used in the specification. Because the specification already has named the single realm parameter (**`Vectors`**) needed for the component, the body file can use references from this realm (e.g. the type clause for the identifier **`Right`** in the **`"*"`** function with such references marked by a pair of brace characters (**`{}`**). Most of the content of a Component Body will appear as ordinary Ada and the EDGE/Ada tools depend on this form to lessen the translation overhead to produce

generated Ada from the body. Any component-for-realm parameter substitution is handled by the toolset which also checks for consistency among the realms and components that are used with one another.

## When to Start

- Necessary COMPONENT SPECIFICATIONS complete. As explained above, besides the COMPONENT SPECIFICATIONS for the COMPONENTS BODIES under development, any realm parameters for components under development must also have Component Specifications in place to allow for code generation and testing. The ASSET DEVELOPMENT PLAN should be followed in deciding how to sequence component developments so that those of highest customer interest are completed first. For components that build on, or are patterned after, other components, plans should suggest how these successive developments can be staged to best support overall asset base development.

- Testing and integration plans understood. While a complete plan for the entire asset base need not be in place before significant effort is applied to COMPONENT BODY development, plans must be sufficiently complete and understandable to the domain engineering team responsible for component development. Such plans are particularly important when there are complex chains of reference among many of the components and realms in the asset base. Testing and integrating one part of the asset base may in turn require sufficient results available from another part. Planning can make sure that these dependencies are clear and a road map is in place to address them.

## Inputs

- COMPONENT SPECIFICATIONS. Both the specifications for the current Component Bodies and for components in realms that parameterize these components are required.

- ASSET BASE ARCHITECTURE. Components are not meant to stand-alone. Their dependencies on other architectural elements are formally established in the ASSET BASE ARCHITECTURE.

- EXEMPLAR SYSTEM ARTIFACTS. Because DAGAR COMPONENT BODIES are written in extended Ada that on a statement-by-statement level rely heavily on Ada syntactic expressions, segments of COMPONENT BODIES can often be created by following and adapting existing high-level language code, especially if the code is available in Ada. If design and requirements materials exist for exemplar materials, these materials can also be applied directly to component design

- ASSET DOCUMENTATION AND TEST MATERIALS. Test materials and documentation abstracts and templates will have been produced during initial architecture development, primarily during the development of the REALM SPECIFICATIONS for the components now under construction. These materials can be used as the basis for the more detailed and individualized material required during COMPONENT BODY development.

## Controls

- ASSET IMPLEMENTATION PLAN. This plan covers the sequence of component development and the strategy and methods to be followed in working effectively as a team to accomplish the goals outlined in the plan. One aspect of the plan will address whether a relatively complete COMPONENT SPECIFICATION effort is followed by comparably complete COMPONENT BODY effort, or whether a more evolutionary development pattern is to be pursued.

## Activities

➤ Write each component body.

The *Develop Component Bodies* task continues that began at the COMPONENT SPECIFICATION level. Attention must now expand to include the whole of *how* things get done and in particular must address algorithmic details that describe how the results and services allocated to the realm are accomplished in necessarily fine detail. The techniques used to complete Component Bodies will borrow heavily from the organization's usual code development practices. As usual, it is best not to jump immediately to the body coding level but to carefully consider the logical results and operations required at the implementation level and then to create implementation code that correctly interprets and carries out the logical steps and produces the expected results. The DAGAR process does not prescribe detailed practices to be followed here.

> Example. Exhibit 34 shows a partial body for the matrix realm component that is an implementation of the **`Std_3_Matrices`** COMPONENT SPECIFICATION shown in Exhibit 32, which in turn elaborates the REALM SPECIFICATION shown in Exhibit 24. The exhibit clearly shows the Ada-like quality of the detailed steps contained in a COMPONENT BODY. With the exception of the use of the realm parameter **`{Vectors}`**, and the component name declaration at the top, all of the lines in the exhibit are standard Ada 83 statements. Notice that the COMPONENT BODY has complete access to the REALM SPECIFICATION holes that are filled in the **`Std_3_Matrices`** specification; e.g., details about the **`Matrix_Index`** and **`Matrix`** types declared in the specification are used in the body.

➤ Test each component body.

Testing of COMPONENT BODIES independently of their integration into client applications is also required. Testing of COMPONENT BODIES cannot begin until at least a few of the realm services declared in the REALM SPECIFICATION have been defined within the component body. See Exhibit 45 on page 107 for an example of the Ada code body produced for **`Std_3_Matrices`** COMPONENT BODY. Once code generation at the component level is possible, the domain engineer can apply the available test material to determine both functional and behavioral properties.

While some stand-alone testing is a necessary, DAGAR places a strong emphasis on architecturally based testing. Such testing requires that the architectural elements connected to the component (realms it depends on, completed components for such realms, etc.) be available and the necessary testing apparatus and materials for these related components must also be available. The architecture will typically contain special realms and components within realms that are defined solely to support the adequate testing of the architectural elements of interest to asset base customers.

## When to Stop

- Asset Base implementation thread complete. While the entire architecture need not have reached the status of completed and tested Component Bodies, a sufficient thread of capabilities of the sort required by asset base customers must be completed before this task can be declared finished. The task will continue with some level of effort applied until the entire asset base, as required to support the currently identified set of asset base customers, has been completed. Individual component tests are followed by increasingly complete integration tests that consider operations of the generated subsystem in contexts close to those expected to exist in applications fielded by the asset base customer.
- ASSET IMPLEMENTATION PLAN goals reached. The purpose of this plan is to guide the domain

engineering team in its production of the asset base. The plan will include clear and quantifiable goals about various states of asset base completion. Attainment of these goals should be viewed as the best set of stopping criteria.

### Guidelines

Essentially, all of the guidelines presented for the *Develop Component Specifications* task are relevant here. They must be interpreted to extend to the level of algorithmic descriptions, but their recommendations generally carry over and require no further elaboration. The engineer should once again be cautious about difficulties encountered during Component Body definition immediately causing reconsideration of architectural decisions made in previous phases and tasks within DAGAR. There can be cause to retract and modify earlier decisions but engineers should remain focused on completing implementation steps and using the proper feedback channels to suggest possible later changes.

## 6.2.3 Develop Asset Documentation and Test Materials

During *Implement Assets*, the domain engineer is designing and implementing the core of the asset base. Along with *Develop Component Specifications* and *Develop Component Bodies*, the engineer must address the task of developing supporting materials that describe what these asset elements do and how they can be tested to verify that they perform as described. Rather than wait until the core development tasks are completed, it is advisable that the developing support materials task should be carried out at the same time as the main development activities.

ASSET DOCUMENTATION AND TEST MATERIALS will extend and particularize the documentation and supporting test material that was developed during *Architect Asset Base*. Since Realm Specifications were completed during architecture development, insufficient detail was known about the components to describe actual test cases or test case results when these tests are to be applied to individual components. At most, black box test plans can be formulated to identify behavioral commonalities and variabilities that arise among the known components that were identified as part of architecture. Similarly, template documentation can be written that summarize activities and features of the architectural modules, but the insides of these templates cannot be written until full component development is well underway. When an application engineer determines which components from the asset base are required to fit an application, and the asset base architecture guides the engineer in the process of configuring and combining them for use, the appropriate supporting materials must be delivered to the engineer to complement the configured components and help guide the engineer to integrate and test them in their actual usage context.

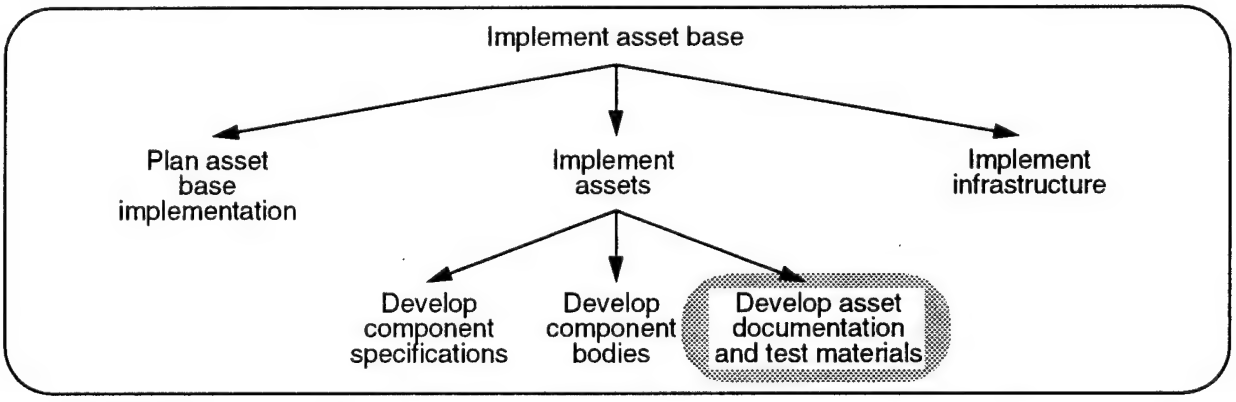



**Exhibit 35.** Develop Asset Documentation and Test Materials Process Tree

The primary purpose of the *Develop Asset Documentation and Test Materials* task is to produce the material as described in the name of the task. These materials are produced for every component and are used during asset development to verify correctness and performance of the component. The materials are also used by the application engineer during application development.

Even though the production of these supportive materials is called out as a separate task in the DAGAR process, this task should not be viewed as a stand-alone task. While staff with special skills (e.g. in software quality assurance) may be called in for this task in particular, they will not be working in isolation from main development activities. There is a significant management challenge that must be met during the Implement Assets phase to make sure that all subtasks of the phase, especially this one, are carried out with plentiful communication and cooperation among team members assigned to the individual tasks.

## Approach

The domain engineer starts with the architecture-based materials and works to refine and extend them. The documentation includes documentation for the application engineer and user documentation that can be tailored by the application engineer to fit the purposes and audience for the application. A primary source of both documentation and test materials is legacy system artifacts. Even though the domain architecture may be the source of a substantially new implementation of application functionality, this new implementation may be used in application circumstances that have been largely unchanged from those surrounding earlier versions of the application. As such, useful material for documentation and testing purposes may already exist among the domain information sources gathered during descriptive domain modeling. Where significantly new functionality is added, or innovative algorithms are designed, the corresponding test and documentation material will have to be developed by the engineer. The usual development methods for such material that is in place within the organization can be applied in producing this new material.

## Workproducts

- ASSET DOCUMENTATION

Documentation should be produced in formats that are already in place within the development organization. Supporting material meant for use by application engineers in widely differing usage contexts should actually be made available in a number of different formats. Many documentation development systems allow for the export of material in a number of standard formats that can then be imported to, and accessed from, tools available to the application engineer.

- ASSET TEST MATERIALS

Test materials include test plans, test cases, test harnesses, expected test results and test result comparers to compare actual results to expected results. Test harnesses will often take the form of special components that are developed as part of the architecture. Such components can be allocated either to existing realms or be placed within special test realms inserted within the architecture to explicitly support asset base testing. Such test realms are not meant to contain components expected to be of use to an application engineering customer except to support other segments of the architecture. The initial identification and positioning of these realms within the architecture should be done during *Architect Asset Base* but further elaboration of them will occur during the *Develop Asset Documentation and Test Materials.*

### When to Start

- Start of *Implement Assets*. As noted above, this task is best approached as an on-going task as component development proceeds. So even while the planning task begins, work on production of the necessary supporting materials can get started with gradual expansion of the activity as more intensive component development gets underway.

### Inputs

- EXEMPLAR SYSTEM ARTIFACTS. Exemplar artifacts can often provide valuable raw material for the creation of asset base supporting materials. Prior documentation sets and test data can prove particularly valuable.
- ASSET BASE ARCHITECTURE. All testing and documentation should be done from an architecture-centric point-of-view. In particular, there may be some designated realms and components in the architecture that are intended to be used as testing scaffolding.
- ASSET BASE ARCHITECTURE. The ARCHITECTURE DOCUMENTATION AND TEST MATERIALS were prepared at the architectural level and therefore will need to be adapted and extended to support component-level testing. A significant amount of integration testing material can be used as-is from that developed during architecture definition.

### Controls

- DOCUMENTATION AND TEST PLAN. The *Plan Asset Base Implementation* task produced several planning documents with this one expressly supporting development of supporting materials. This is the primary control on this task.
- COMPONENT SPECIFICATIONS. The nature and degree of testing materials that are required depend strongly on the components themselves. One could argue that COMPONENT SPECIFICATIONS (and COMPONENT BODIES) are best viewed as inputs to this task.
- COMPONENT BODIES. Could also be viewed as input.

### Activities

Documentation and testing should be done hand-in-hand with the development of the asset base itself. The purpose of this task in DAGAR is to highlight its importance in producing high-quality assets. Unfortunately, the EDGE/Ada toolset does not yet provide any enactment mechanisms or work product support for the creation of either documentation or test materials. These must be created and stored using the organization's available software engineering environment services.

➤ Extend Architecture Documentation.

In all aspects of DAGAR-based domain engineering, the domain architecture is the key to producing usable and understandable results. Documentation activities should begin with a review of the available documentation from the Define Asset Base Architecture sub-phase. Now that components are under development, many of the holes and opportunities defined at the architectural level are being filled in with real code and real implementation choices. The documentation at the component level can be viewed as describing how these holes were filled and the opportunities realized. If component development reveals flaws in the architectural design, a feedback loop in the DAGAR process will provide the opportunity to correct these flaws and the corresponding architectural level documentation must be corrected as well. Since documentation will primarily

exist in template form using the locally available documentation mechanisms, engineers will use documentation processing tools to open the templates, manually insert the component-dependent elaboration of the template, and the save a copy as the actual document output of the activity.

➤ Review exemplar artifacts for possible source material.

While DAGAR domain engineering is based on particular specification and implementation languages supported by the EDGE/Ada toolset, detailed approaches and implementation concepts will often be suggested by previously implemented systems. If these systems were produced using accepted development practices, there will be significant supporting material that was saved as part of the legacy materials from these efforts. Just as the design and implementation of parts of these systems can be used to pattern component designs and implementation, the supporting materials can also be used. If the legacy documentation exists in an electronic form compatible with that being used to support DAGAR, it can be processed electronically and provide valuable starter and supplementary material. Similarly, legacy testing materials (especially test plans and test data) can be used as stepping stones to producing testing material at the component level.

➤ Build on architecture testing framework.

Testing is vital to the success of any implementation endeavor. As such testing was considered early in the architecture development effort and a task in DAGAR is devoted to producing testing materials as part of the architecture. In particular, a well-designed DAGAR asset base will include at least one, and perhaps several, realms in which the presence of components solely designed to support the testing of other parts of the architecture are indicated. Testing activity, and the production of testing materials, should be based on the use of these components. Early attention should be paid to the implementation of such components and they should be used often to execute testing scenarios based on test data developed during this task. Production of new testing materials should stress effective use of these testing parts and provide data that supports understanding how components compare to one another when accessed from these test components.

### When to Stop

- Component documentation sufficiently complete. Stopping the documentation process will coincide with the stopping of the component development process. If these activities are synchronized with one another, documentation has been proceeding apace with component development. If schedules or staffing concerns have led to these activities being de-coupled, attention must be given to reviewing documentation status periodically to insure that and disconnects and inadequacies are addressed.

- Testing materials in place to support application engineer. Asset production processes will regulate the application of test materials internally to asset development. Testing occurs on a periodic basis and materials must be in place to support this testing. The domain engineer must also work to produce testing materials of use at application development and integration time. This means that some understanding of expected usage contexts is required to prepare appropriate testing materials. The *Develop Asset Documentation and Test Materials* task must continue until the foreseen usage contexts are supported by test materials that can be used in these contexts. Since later usage profiles may appear, it may be necessary to return to the task to expand the test materials accordingly.

### Guidelines

- Follow incremental pattern. The production of supporting materials should follow the same pattern as that followed during the production of COMPONENT SPECIFICATIONS and COMPO-

NENT BODIES. Echoing the advice given in these task descriptions, the domain engineer should plan the construction of the asset base so that a thread through the asset base is produced as early as possible and that additional layers and more completely functional components are added as component development proceeds. As these initial components are created, the supporting material for these components should be created along with them rather than be postponed until the end of the implementation material. Once basic materials exist, they can be refined and extended according to the changes made to the components themselves.

- Ensure sufficient staff communication. While a small project may be run with only a small number of engineers performing all activities in conjunction with asset base implementation, for larger projects, staff size increases can present additional problems that need to be worked out. One of these problems is that staff members will often have specialized duties and expertise. Careful hand-offs must be arranged as these duties overlap. The production of documentation and testing materials are often allocated to specialists in these areas and so as development of component drafts and versions occurs, these products must be handed over to those with expertise in documentation and testing. Communication among staff members needs to be facilitated and interim documents such as notes, trouble reports, informal test data, etc. needs to be made available as raw material for the *Develop Asset Documentation and Test Materials* task. EDGE/Ada as currently configured does not offer significant support in this area, so other development platform services need to be selected and applied in support of communication.

## 6.3 Implement Infrastructure

The word ***infrastructure*** is used in this task to describe the technology supporting DAGAR itself, including all of EDGE/Ada. Individual domain engineering projects and programs are expected to identify changes and extensions to this technology and with the availability of the source code for EDGE/Ada, the opportunity exists for implementing these changes. Such modifications represent one aspect of infrastructure. Of particular importance to the *Implement Asset Base* phase are those elements that support usage of the asset base by customers. None of this infrastructure needs to be developed from scratch.

The other aspect of infrastructure meaningful to DAGAR are the data structures that must be developed during domain engineering to support the use of the asset base during application engineering. These data structures are presented through EDGE/Ada to guide application engineers in selecting and configuring assets from the asset base. Some of the data structures are actually generated by EDGE/Ada — for example, an RLF representation of the asset base architecture is produced automatically from the ARCHITECTURE SPECIFICATION. Another important data structure contains the composition rules that record semantic considerations of which components can be meaningfully used together with other components. These rules are created on a component by component basis by the domain engineer as component development continues to the point of supporting explicit asset base customer usage contexts. The current version of EDGE/Ada does not fully support the authoring of these rules, although work is planned to improve matters in this area. EDGE/Ada uses the NASA CLIPS rule-based inferencing system to process these rules and the domain engineering team should identify a staff member to learn CLIPS operations and the corresponding rule expression language supported by CLIPS.

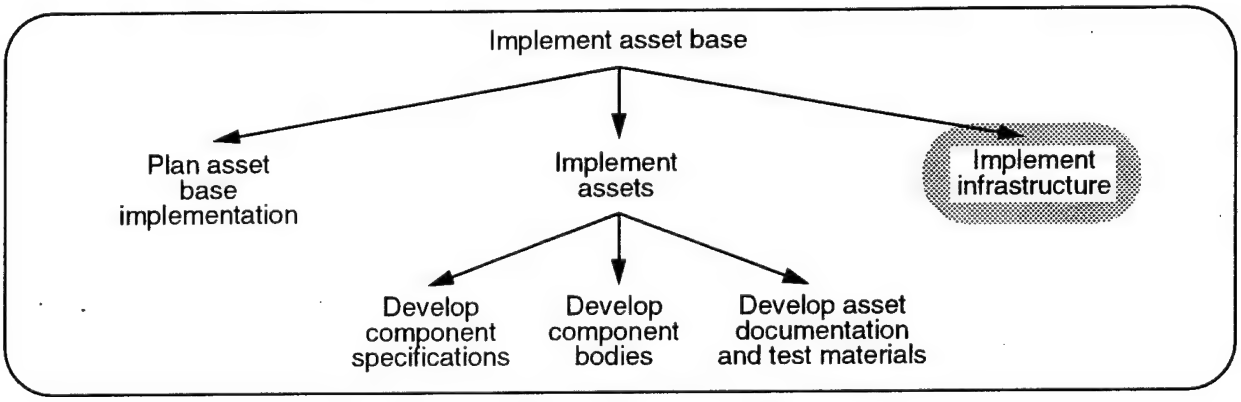



**Exhibit 36.** Implement Infrastructure Process Tree

The primary purpose of the *Implement Infrastructure* task is to produce the data structure elements that underlie EDGE/Ada when it is being used by the asset base customer, and to identify, and possibly implement, modifications to the infrastructure. It is expected that every domain engineering project that uses DAGAR will need to at least develop a set of component composition rules that are used during application engineering. Many, if not most projects, will not extend the EDGE/Ada toolset, but may identify suggested improvements and report problems experienced while using the toolset. Some projects may go further and add capabilities into the toolset or modify some of its components to better reflect organization goals and methods. One already identified opportunity for improvement covers the use of the domain models created during period leading up to architecture development as a means of cataloguing and providing access to the components contained in the asset base. The domain model was used directly in defining the architecture but is currently not made available for during asset utilization by EDGE/Ada.

In general, it is difficult to look ahead to the time when someone other than the developer will be using the products of domain engineering during application engineering. It will be hard to anticipate the problems and integration concerns that will become evident as elements in the asset base are applied in a "real" application context. Nonetheless, this kind of anticipation will greatly aid the utility and eventual success of the asset base and is an important part of the DAGAR approach. Any lessons learned from the first users of the asset base should be fed back into the *Implement Assets* phase and this feedback will particularly be appropriate during the *Implement Infrastructure* task.

While there may not be many projects that will decide to make their own extensions to the EDGE/Ada toolset, those that do need to be aware of the impact that making and integrating these extensions may have on the overall implementation of assets. The STARS Demonstration project practiced an early form of DAGAR where essentially all of the EDGE/Ada toolset was being developed and delivered for use on the project concurrently with the production of assets. This simultaneous development and application of infrastructure caused some friction among developers and users and had an impact on productivity. Careful management of use and modification of the infrastructure can alleviate these concerns to some degree but even with the best management approach, the domain engineering project should expect to seem some impact while infrastructure changes are being applied. Rather than attempt such infrastructure changes concurrently with application, changes can be identified and implemented, but not applied in the current development context. They can instead by applied in the next project, or at least in a distinct later phase of the current project where a sufficient transition period is scheduled to mitigate transition concerns.

## Approach

Infrastructure to support the use of the asset base by application engineers is developed along with the assets. In the DAGAR approach, a significant portion of this infrastructure has already been created in the EDGE/Ada toolset. EDGE/Ada includes provisions for two ways for application engineers to select assets from the asset base. The first method involves direct selection of components using GLUE (Graphical Layout User Environment). GLUE was originally developed by Loral Federal Systems (now part of Lockheed/Martin) in support of their GenVoca-based design for an avionics domain within the Domain Specific Software Architecture program[3]. GLUE has been incorporated into the EDGE/Ada toolset and the domain architect need not develop any special support for the use of GLUE since it is automatically driven directly off of the architecture.

The second piece of infrastructure provided for selecting assets is the Architecture Configuration Assistant (ACA) tool that is included in EDGE/Ada. To support application engineers in their use of the ACA, the domain engineering team creates CLIPS asset composition rules that determine which assets can be combined with others in producing products from the asset base. While all components in the same realm are compatible at the interface level (so-called plug compatibility), they are not necessarily semantically compatible when used with other components in other parts of the architecture. By writing rules expressed in the CLIPS rule-based language, the domain engineer is able to declare configuration constraints that reflect how assets may be used in combination with each other. The creation of these CLIPS rules is the only part of the asset base infrastructure that cannot be automatically generated using EDGE/Ada. These CLIPS rules are used in concert with an RLF representation of the domain architecture (generated automatically by EDGE) by the ACA to determine choices that the application engineer can make, based on the choices already made.

Additional opportunities for customer support may also be identified during this task. Rather than require use of the ACA, other more informal access methods may be provided, at least for expert users. Using a tool such as the RLF directly to create a model of the asset base, RLF access mechanisms can be used to search for assets of interest and provide direct retrieval of their contents.

However, due to the generative nature of DAGAR, the use of the EGDE/Ada toolset will still be required to produce Ada language modules for integration into the customer's intended application. Providing for semi-automated feedback mechanisms from asset base customer to asset base developer would also prove very useful for long-lived domain engineering efforts.

## Workproducts

- ASSET BASE INFRASTRUCTURE

It is more likely that recommendations for improvements or corrections to the existing infrastructure will be produced during this task rather than extensions or modifications themselves. DAGAR is supported by EDGE/Ada which features built-in support for application of the DAGAR method. Specific extensions to EDGE/Ada may be suggested based on other software engineering environment capabilities that are expected to be in place within an asset base customer's location. For example, any reuse library mechanisms already in use within a customer's organization may need to be accommodated and perhaps integrated with the infrastructure provided by EDGE/Ada. Additional general purpose modifications to the EDGE/Ada environment may be identified and in rare cases implemented.

The ASSET BASE INFRASTRUCTURE also includes asset constraints. Asset constraints encompass the component composition rules that have been discussed previously. These rules are expressed using an appropriate specification language as required by the EDGE/Ada toolset component used to process these rules and interact with the asset base customer. The current version of EDGE/Ada uses CLIPS in an integrated fashion with the RLF for this purpose. As a result, the rules must be expressed using CLIPS conventions and notation.

## When to Start

- ASSET BASE MODEL complete. The ASSET BASE MODEL provides information about the target customers for the asset base as well as a profile of the Asset Base itself. This information will is needed to match customers, and the known asset usage contexts for these customers, to the asset base. Application development methods employed by the customer will also be used to identify infrastructure enhancements that may better integrate the production of assets with their use.

- Sufficient component implementations complete. In order to express constraints to be enforced during component integration and composition, fairly detailed knowledge about component implementation must be available. Some constraints may be known once the COMPONENT SPECIFICATIONS have been developed, but before COMPONENT BODIES are completed. It will be more common that only after bodies have been completed, or at least prototyped, can the full extent of component-to-component interoperability be understood. While the entire set of components need not be complete before this task begins, a representative cross-section must be available.

It should be noted that postponing this task until the entire contents of the asset base is completed is not advisable. As is the case for other parts of DAGAR, an evolutionary spiral approach where increasingly richer variants with more complex interactions follows a relatively sparse and spartan implementation activity is preferred. Even during an early arm of the spiral, asset composition rules should be considered and early versions of these rules integrated with that part of EDGE/Ada that processes them.

### Inputs

- ASSET BASE MODEL. As noted above, this model provides vital customer information
- ASSET BASE ARCHITECTURE. Components are composed in terms of the architecture and therefore composition constraints must reflect and respect the architecture.
- ASSETS. Assets are the realms, and more importantly the components (realm implementation variants), that make up the asset base. Assets may also include key documentation elements and test materials that help understand the extent of component composition requirements and limitations.
- COMPONENT SPECIFICATIONS. These are the asset base elements that detail how interface elements presented in a REALM SPECIFICATION are made concrete enough so that translation to an Ada interface specification is possible.
- COMPONENT BODIES. Bodies provide the detailed how-to information necessary to generate the Ada code that will finally implement the services published through the realm interface.

### Controls

- INFRASTRUCTURE IMPLEMENTATION PLAN. The *Implement Asset Base* phase calls for the production of this specific planning document to guide this task. Among other elements, it will outline scheduling constraints on the presence and completeness of infrastructure elements supportive of planned asset base evolution and release activities.

### Activities

As suggested in the work products identified as coming out of this task, there are two main activities comprising this task. The overall task requires understanding what needs to be done and then doing it within the EDGE/Ada toolset.

➤ Identify Infrastructure changes and extensions

While most projects will not actually implement infrastructure changes, all projects should at least consider improvements that could make future projects that follow the current one more efficient and of higher quality. While this task is most directly focused on asset implementation and supporting asset utilization, all aspects of the architecture instantiation and application process should be considered. There is no specific work product template. A simple text-only or word processor document can be used for this purpose. If any of the identified changes or extensions are required for successful completion of the current project, they should be highlighted and communicated to project management as soon as possible.

➤ Implement Infrastructure modifications

When sufficient expertise exists, and project resources and schedule permit, proposed extensions and modifications to the actual asset base infrastructure can be implemented in this task. Access to the EDGE/Ada toolset in source code form will be necessary to make substantial changes. The GUI wrapper for the toolset can be extended to add access to additional tools and capabilities. Since the RLF includes an API, those parts of the toolset with RLF dependencies can be modified by working from the API. The CLIPS component has its own set of extensible interfaces that can be used to add features or functionality. These changes should be managed using the normal software development practices in use within the organization.

➤ Identify component composition dependencies and conflicts

Before the asset composition rules can be formalized, they must be first conceptualized and made specific to the components and realms present within the architecture. There are two main kinds of rules. One expresses the condition that if a certain component is selected for use on one part of the architecture, it (or a specific component depending on it) must be used in another part of the architecture. A special case of this condition occurs in configuring an instance of the architecture that makes use of a realm with only one component. In this case, there is no need to force the user to select that one component — it should be selected automatically. Similarly, any "select this component if that component is selected" rule should have the consequence applied automatically.

The other sort of rule concerns forbidding a component (or components) from being used in the same system instantiation that requires the use of one or more desired components. Here, the presence of one or more components precludes the presence of others. In the case that the application engineer is selecting components for a subsystem, picking one of these trigger components should cause the correspondingly forbidden components from being offered for selection. Both of these kinds of rules are expressible using the EDGE/Ada CLIPS-based capability. Before they can be so expressed, they must be recognized and recorded informally.

➤ Record composition rules using rule specification language

Once the asset constraints have been determined informally, or at least a sufficient portion of them, they must be converted to a form processable by the EDGE/Ada toolset. The ACA (Architecture Configuration Assistant) is currently based on CLIPS and the constraints must be specified using rule syntax supported within CLIPS.

➤ Test effectiveness and correctness of rules within Architecture Configuration Assistant

Once the rules have been recorded formally, they need to be tested in the environment within which they will be used by the application engineer/asset base customer. This testing can take place by involving real customers and arranging for them to "beta test" the configuration framework and record their findings. If an actual asset base customer is unavailable, members of the domain engineering team will need to adopt the role of customer and carry out testing activities in a simulated customer usage environment. The ACA will be used to process the rules and carry out subsystem instantiation steps that conform to the rules. The end result should be a workable subsystem that respects all rules that apply during the customer's interaction with the asset base.

### When to Stop

- Asset constraints sufficient to support Asset Base use. Since the goal of this task is the production of infrastructure materials to support hands-on access to the asset base, this task must continue until these materials are sufficient to guide application engineers in retrieving and configuring assets that can then be integrated with other application code. Unless this material is of sufficiently high quality, asset base customers may develop negative opinions about the usability of the asset base and thereby affect the pay-off that was expected of the asset base. There can continue to be on-going work in improving the level and completeness of the usage support offered through the asset constraints, but they need to achieve a level of completeness that productive access to the asset base can be assured.
- Infrastructure modification requirements recorded for possible implementation. While most projects will not actually make significant modifications to the infrastructure itself, a project will likely note operational problems and opportunities for improved support within the infra-

structure. The Implement Infrastructure task should not end until a written summary of these observations is produced and forwarded on to those members of the organization with responsibility to make changes to the infrastructure.

## Guidelines

- Involve real application engineering customers. Since the results of this task are meant to support staff members other than the asset base developers themselves, it is advisable to seek an early "friendly" application engineering customer who can be relied on to give honest and insightful opinions of the usability of the infrastructure and who can tolerate some degree incompleteness and robustness in the infrastructure. Such a customer may be hard to locate, but if one can be found, there can be measurable quality and schedule improvements as a result of the interactions with such a customer.

- Test infrastructure using non-developers. Failing to find an early customer fitting the description given in the previous guideline, the domain engineering team should use internal staff members who are not intimately familiar with the details of component implementations and connectivity as a means to conduct a fair and complete test of the infrastructure. It will be easier for such staff members to adopt a stance that accurately reflects asset base users and who can express reservations about the interaction they have with the asset base through use of the infrastructure.

# 7.0 Apply Asset Base

The *Define Asset Base Architecture* phase of the DAGAR process produced a formal architecture specification along with the specification of a set of major modules (called realms) into which the major areas of functionality to be provided by the asset base have been partitioned. The ASSET BASE ARCHITECTURE lays out what the key architecture variants are — these are provided by the components that implement the functionality defined by each realm — and depending on how these components are combined, various system architecture possibilities can emerge from the asset base architecture. The realm descriptions define the services and objects that are encapsulated within the realms themselves.

The *Implement Asset Base* phase of the DAGAR process completed implementations of the DAGAR components which exist within each realm. In DAGAR, such component implementations include code fragments for substitution points defined in the realm declarations and complete specifications, using an extended Ada syntax, for how the services encapsulated in the realm are accomplished. Besides the components themselves, the asset base implementation phase also produced supporting materials such as test plans, documentation templates, and architecture usage rules that help an application engineer develop subsystems in terms of the architecture. These rules are used by the EDGE/Ada toolset in providing system composition guidance to application engineers.

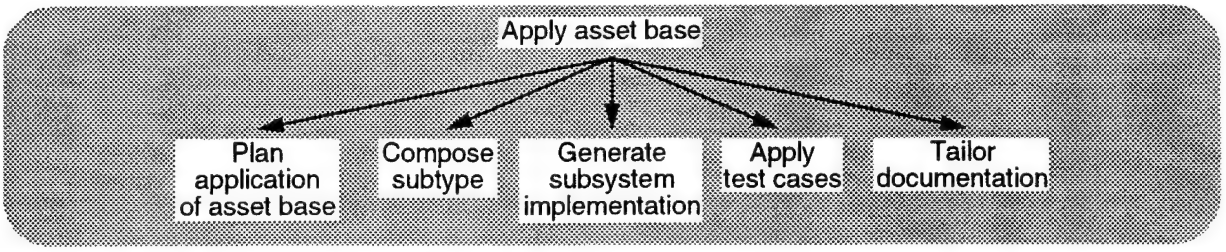



**Exhibit 37.** Apply Asset Base Process Tree

The *Apply Asset Base* phase of the DAGAR assumes that an asset base of sufficient maturity exists that engineers building applications that require services available through assets in the asset base can effectively select which assets are appropriate in a specific application context. Support for asset examination and selection is provided through the asset base infrastructure, maintained and presented through the EDGE/Ada toolset. The purpose of this section of the DAGAR process guidebook is to examine how this support is provided and to define the steps of an application construction process that is informed and guided by this toolset. Note that this portion of the DAGAR process will often be performed by different team of engineers than those who designed and built the asset base.

Just as training is required for the domain engineering staff, application engineers must also receive training to establish proper expectations regarding how to obtain assets and how to use these assets in combination with the rest of the system being constructed. ODM provides support for the creation of small, "horizontal" domains that provide part of the functionality required to field a particular system. As such, assets produced for these domains provide partial coverage for the set of services that comprise the application system. Therefore, a certain amount of integration and system-level, in-place testing must take place after the assets are selected. While the ACA and GLUE components of EDGE/Ada help the engineer to select and integrate the assets with respect to the architecture, the application engineer must work within the application system development environment to perform the final steps needed to complete application system development, including integrating assets from the domain with the rest of the application.

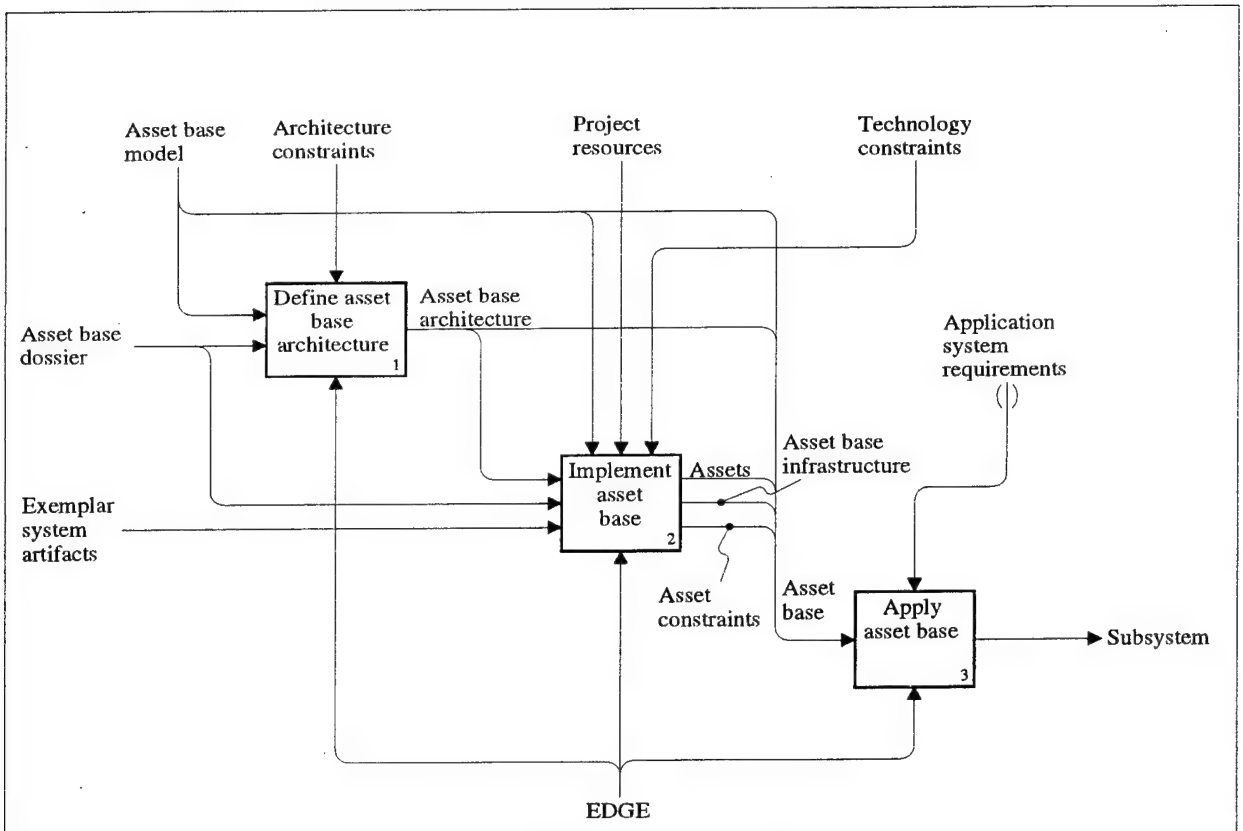



**Exhibit 38.** Apply Asset Base $IDEF_0$ Diagram

## Approach

Once the architectural framework and asset base is in place, the task of applying the asset base in the context of the framework can begin. Asset base application includes planning asset base utilization, composing a subsystem from the available assets (with help provided by the EDGE/Ada toolset), generating Ada source code that implements the system (again with the help of EDGE/Ada), applying test cases, and finally tailoring asset base documentation for use in the current application context. The asset base infrastructure will assist application engineers in selecting assets and will automatically constrain dependent selections where possible. If there is only choice to be made in the context of other choices already made, that one choice will be made automatically. Some of the sub-tasks of Apply Asset Base are not currently supported by EDGE/Ada (e.g., the planning and documentation tailoring sub-tasks). These must be conducted using other mechanisms available to the engineer through the organization's software engineering environment.

Exhibit 38 contains a summary of the required tasks making up this phase. The main goal is to compose a subsystem and then generate the subsystem implementation. The subsystem is then manually integrated into the rest of the application system. Feedback to domain engineers about the utility of the asset base architecture and assets should occur throughout asset base application. Based on this feedback, improvements are possible to assets and their configuration with respect to other assets. Through the generation facilities of EDGE/Ada, these changes can automatically lead to new Ada code ready for integration into the application. The application engineering team can be informed automatically of subsequent corrections and changes that take place in the asset base. If these changes do not directly impact the current application, they can safely be ignore. Otherwise, a configured set of assets based on the last set of selections made by the application

engineer can be created automatically. Manual re-integration with the rest of the application will still be required but if this activity was carefully documented during its first iteration, the amount of additional work will be minimal. All applications of the asset base architecture and assets should be used to feed back lessons learned and support improvements to the ASSET BASE ARCHITECTURE and ASSETS.

## Results

With a sufficiently robust asset base architecture and assets, the application engineer is able to concentrate on application-specific needs and is still assured of obtaining tailored code for use within the application. Some previous work on reuse and reusable components conveyed the impression that reuse was normally obtained through the use of general purpose components that were engineered expressly for the purpose of supporting multiple contexts of use. To do this, the components often had to be defined with a large number of internal capabilities, many of which were not applicable in a particular context. This extra bulk introduces needless complexity and often inefficiency into an application. DAGAR affords the application engineer the opportunity to make effective choices among available functional and performance characteristics and to have Ada code generated that provides the needed capabilities without unnecessary features and unused code.

Successive applications of the domain can be used to recover costs associated with building and maintaining the domain asset base. In setting up a product line approach to development and application of software assets, an organization must be concerned with how to pay for the creation of the domain and its subsequent maintenance. Careful monitoring of how and when the asset base is consulted and used can lead to a cost sharing and allocation scheme as parts of the organization begin using of assets contained within the asset base.

## Process

As shown in Exhibit 38, the *Apply Asset Base* phase has been decomposed into five sub-processes:

- In the *Plan Application of Asset Base* task, application engineers consider the surrounding circumstances that may affect their ability to make use of assets selected from the asset base. If any significant constraints arising from these circumstances were known to the domain engineering team at the time the architecture and assets were being developed, it should be relatively straight-forward to see what application areas (or sub-areas) are covered by the asset base and make plans to use the asset base to cover these areas.

- The primary goal of the second task, *Compose Subsystem*, is to make application-sensitive selections from among the components present in the asset base. Both the Architecture Configuration Assistant (ACA) and the Graphical Layout User Environment (GLUE) tools within EDGE/Ada are available to help in accomplishing this goal.

- In the third task, *Generate Subsystem Implementation*, application engineers again use EDGE/Ada to generate the Ada code that can then be integrated with the rest of the application.

- In the *Apply Test Cases* task, application engineers extract the appropriate test materials that support the assets selected in the previous step and apply them in a working or simulated application context. The test cases that are available need to be applied in the same context in which the code generated from the assets must execute.

- The fifth task, *Tailor Documentation*, also starts with some baseline material developed during domain engineering that describes the abstract properties and behavior of the assets available from the asset base. Application engineers now must make this information concrete and reflective of how the assets are going to applied in the current application. If the application context was largely anticipated during domain engineering, this task may not require significant resources to complete.

## Sequencing

- Tasks in the *Apply Asset Base phase* lend themselves to sequential execution. Contrary to most of the other DAGAR phases and tasks, the five tasks of Apply Asset Base can be planned to occur in more or less sequential fashion. The compose and generate tasks will typically be performed in an integrated fashion, but even here, a complete composition activity can be performed prior to the first generation of Ada code. After code is available it can be tested. Documentation tailoring does not need to wait until testing is complete, but there is no essential harm in waiting, especially if some asset selections are tentative subject to testing results.
- Impact of Generation-centered application development. DAGAR features an integrated use of an Ada code generation capability. In most cases, the context of use for the generated code will be in combination with handwritten Ada code. This integration may require the application engineer to at least visually inspect the generated output when considering integration-related issues or problem solving. If the generated code requires modifications in support of integration, the best procedure to follow is to inform the domain engineering team and have asset base changes made to the assets themselves. The application engineer then re-generates the Ada code to complete integration. If time constraints prevent such changes being made at the source level, some temporary patches can be applied to the generated Ada code files. The application engineering team should be in communication with the domain engineering team to insure that indicated changes are eventually made to the asset base.

# 7.1 Plan Application of Asset Base

Asset base application is part application engineering. Since support exists within the EDGE/Ada toolset for interaction with the asset base, the application engineering team needs to plan for how it can best be served by the products of domain engineering. This section of the DAGAR process covers some issues that must be addressed by the application engineering team in light of the support offered by the toolset and the need to most effectively make use of this support.

Use of DAGAR constrains the method of asset integration with application code that is not part of the asset base. DAGAR itself is based on the use of generation technology (implemented within the EDGE/Ada toolset) rather than manual construction of assets. Planning must therefore focus on how to best integrate generated code components with the rest of the application. While a generation approach might not be suitable for every domain, a significant number of domains can effectively be served with the application of DAGAR and its supportive toolset. When an application intersects with one or more of these domains, significant reuse opportunities exist that can and should be exploited. This planning step insures that project methodology is adapted to best make use of these opportunities.

Another consideration to be made during planning is how the infrastructure available through EDGE/Ada will be provided to application engineers to allow selection of assets from the asset base in a manner appropriate to their application. The EDGE/Ada toolset provides tools to allow the application engineer to automatically select assets based on the asset base architecture. Training in the use of these tools must be budgeted and scheduled.

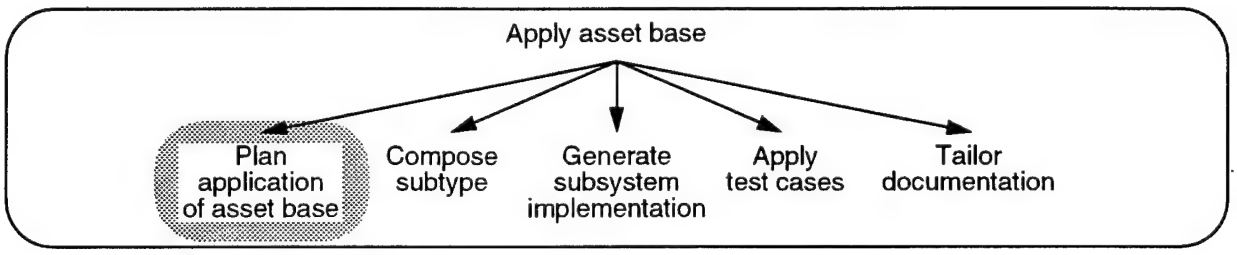



**Exhibit 39.** Plan Application of Asset Base Process Tree

The primary purpose of the *Plan Application of Asset Base* task is to determine how the asset base will be used in the application system, and which parts of the asset base meet system requirements. Integration issues must also be considered for how the assets will integrate with other parts of the system. The plan must consider both what sort of application will be using the assets and when the assets will be needed. If asset base construction is underway at the same time that application development is proceeding, the completion of asset base products can be staged in a way that accommodates domain engineering resources and application engineering needs.

The use of DAGAR features a specific binding of particular asset implementation technology choices to the production of individual assets. Application engineers require sufficient training in the methods of generator-based engineering in order to be successful. By grounding the syntax and semantics of the component specification language in Ada, it is expected that engineers with Ada development backgrounds will find it easy to move between Ada and the use of DAGAR to produce the various parts of the application.

If the structure and content of the asset base is sufficiently compatible with the needs of the application, there should be little if any need to modify the generated code. Any interface incompatibilities should be reported back to the domain engineering team in consideration of changing the source code of the components and/or realms to remove these incompatibilities. The code gener-

ated by EDGE/Ada is Ada in a form that will be readable and close in style and presentation to good quality handwritten code. Planning should address how the application engineering project will handle any mismatches between code produced from the asset base and code that makes up the rest of the application, especially in light of constraints in the application engineering schedule as a whole.

## Approach

There is no support built-in to EDGE/Ada to develop the Asset Base Application Plan. The application engineering team should use whatever standard planning tools are in use within the organization. Concerns to be addressed in the plan include: personnel constraints, training schedule and resource commitment, overall application development schedules, coordination with on-going domain engineering activities, testing of assets in their application context and production of tailored documentation that also reflects the application context.

## Workproducts

■ ASSET BASE APPLICATION PLAN

This plan contains describes the application engineering team's proposed effort to construct at least part of the application using assets and supporting material extracted from the asset base. It contains a descriptive summary of how the basic sub-tasks of *Apply Asset Base* are to be performed within the context of the organization's own strategic plan. Since the resources obtained from the asset base form only part of the application, special attention must be paid to how the asset base material is combined with other application resources. The plan also should address how feedback is given to the domain engineering team and how updates to the asset base are later reflected in changes to the assets that were integrated into the application previously.

## When to Start

- Asset Base judged ready for use. Completion of the ASSET BASE is not necessary prior to beginning the development of the ASSET BASE APPLICATION PLAN. However, the asset base must be sufficiently stable that is possible to understand what functionality and features will be available in the asset base so that planning how much application coverage is possible from the asset base is possible.
- Overall application development effort about to begin. Application development planning as a whole should be complete or nearly so. The application engineering team should be ready to begin concentrated application development where near-term use of the asset base is anticipated.

## Inputs

- ASSET BASE. The contents of the ASSET BASE itself are the most basic underlying information source for the construction of the ASSET BASE APPLICATION PLAN.

## Controls

- APPLICATION SYSTEM REQUIREMENTS. While domain engineering must consider constraints that arise from a number of identified or anticipated customers, application engineering typically is concerned with the needs and wants of a single customer — the one paying for the application under construction. The requirements for this application will drive all aspects of

the application development process, including the intended or potential use of asset base resources.

## Activities

The activities included in this subsection should not be viewed as requiring sequential performance. Nor should the list of these activities be viewed as an exhaustive list.

➤ Assess experience of application engineering personnel

Any plan needs to consider and reflect the potential performers of the plan. In particular, any new technology approach requires people open to the use of innovative techniques and the willingness to learn how to apply these techniques. In preparing the ASSET BASE APPLICATION PLAN, the application engineering team should be evaluated to see what impediments might exist to carrying out the plan.

➤ Plan for DAGAR training activities

For personnel who are identified as being open to the use of DAGAR, but are not yet trained in its use, this training needs to be planned and offered as appropriate. Because of the significant tool support provided through EDGE/Ada, this training is not substantial but nonetheless needs to be scheduled and given prior to serious usage of the ASSET BASE.

➤ Consider impact of ASSET BASE integration with surrounding Application

Where a significant amount of the application under construction is not provided through use of the ASSET BASE, planning must consider how and when integration activities must be performed to produce the final application. If the application engineering team has not had any experience with use of the ASSET BASE, successful integration may present problems so that sufficient time and personnel resources need to be budgeted to complete this integration. Another reason to plan this activity carefully is that the EDGE/Ada toolset does not offer any support for these integration activities. Finally, if the ASSET BASE itself has not had many application users of its resources, there may be problems with the assets that will only be made visible during integration.

➤ Plan interaction between domain engineering and application engineering

An organization will be composed of one or more application engineering teams and one or more domain engineering teams. Successful growth and quality improvement in the ASSET BASES supported by the domain engineering teams assumes that feedback and feed-forward activities are in place within the organization that carry important communications back and forth across the teams. A single application engineering team needs to identify how it can tap into this information flow and provide bug reports and feature requests back to the team responsible for the ASSET BASE being used. The application team also should have procedures in place to receive and act on changes that are made to the ASSET BASE.

➤ Plan integrated testing of assets in application context

Assets will have been tested in their more general asset base membership roles, but will not have been tested and evaluated in the specific application context of interest to the application engineering team. After integration concerns, this testing is the most crucial activity for successful use of the ASSET BASE. Planning should allow adequate time and resources for completion of application-specific testing.

➤ Plan for adaptation and integration of ASSET BASE documentation with other application materials

As identified during integration activities, a number of supporting ASSET BASE materials will need to be obtained and adapted for use in the current application context. The materials that exist for assets in the ASSET BASE will need to be evaluated in terms of their completeness and currency for the application under construction. Again, sufficient resources must be budgeted for adaptation and extension of these materials.

### When to Stop

- ASSET BASE APPLICATION PLAN sufficiently complete. In a very basic sense, the application engineering team can stop this activity when they have achieved what they expected to achieve: a sufficiently complete and understandable plan is in place that can be used to inform, monitor and measure the actual application of the ASSET BASE to the overall application engineering effort.
- Feedback mechanism to domain engineering team in place. Successful amortization of the cost of producing the ASSET BASE assumes successful applications of the ASSET BASE. The asset base will improve only if users of the asset base take the time to provide feedback. It is vital that at the end of the planning activity, each application engineering effort have identified how it will provide this feedback.

### Guidelines

- Inexperienced application engineering teams require more detailed plans. Plan development should be adapted to the anticipated needs of the followers of the plan. If the team performing asset base application is relatively experienced, they will need less detailed plans and less careful monitoring. Conversely, a team with no experience will need a detailed plan with subactivities within the plan carefully explained. The type of plan needs to be driven by the plan's anticipated audience.
- Parallel application and domain engineering efforts should be reflected in integrated plans. Significant opportunities for fertile cross-communication can be lost if communication opportunities are not reflected in the plans.
- Just-in-time planning risky. While the production of a complete plan is not required before the *Apply Asset Base* phase starts in earnest, it is generally a mistake to stagger the production of the plan too tightly with the performance of segments of the plan. Such an approach makes it easy to loose sight of the larger picture of application development and miss opportunities for more efficiently performing application development.

## 7.2 Compose Subsystem

The assets and asset base support system presented through EDGE/Ada are used by the application engineer in the construction of at least a portion of a larger system that is the ultimate goal of a system engineering effort. Typically, the portion of the overall subsystem addressed by the asset base falls at the level of a subsystem whose elements are currently part of the asset base. The infrastructure helps the application engineer determine which assets among those available in the asset base are most appropriate for the system being built and also guides the engineer in composing the assets together in a configuration that can be integrated into the application as a whole.

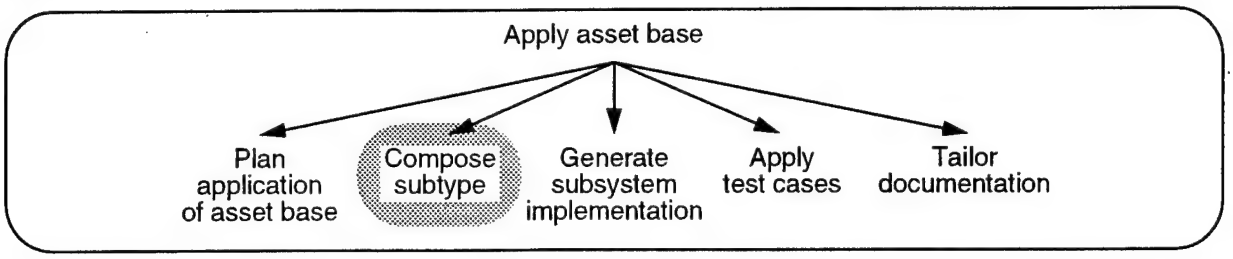



**Exhibit 40.** Compose Subsystem Process Tree

The primary purpose of the *Compose Subsystem* task is to produce a specification of the subsystem in a form that allows the generation of Ada code that can then be integrated into the application itself. There are two primary interfaces provided in EDGE/Ada to the application engineer: a graphical view of the asset base architecture where the choices available at each choice point (realm) in the architecture are depicted in a tree format, and a structured conversation (dialogue) presented by EDGE/Ada where the engineer can respond to questions prepared by the domain engineer that help to determine which assets (components) in the asset base best meet the needs as understood by the application engineer. If the application engineer prefers to use the tree to manually inspect and walk through the architecture, it will be up to the engineer to inspect and learn about the components to choose which of them should be selected for use in the application. The dialogue method, in addition to presenting structured information about the components, also uses the tree view to show how the component fit together in terms of the architecture.

The information presented through EDGE/Ada to the application engineer is only as good and complete as it can be understood by the domain engineers who construct the asset base. The knowledge necessary to do a thorough job in this regard will likely be imperfect at the beginning of period of asset base application. As such, the domain engineering team may need to be active as consultants to the application engineering team during initial applications of the asset base. Knowledge gained during early uses can then lead to significant improvement in the information captured and presented through the asset base infrastructure and thereby enable eventual stand-alone use of the EDGE/Ada tools in support of subsystem composition from the asset base.

### Approach

Through the production of the ASSET BASE APPLICATION PLAN, the application engineering team will have a fairly clear idea of the objects and services available in the asset base and the extent to which these can meet the needs of selected portions of the overall application. A segment of the application engineering team will receive any necessary training in use of EDGE/Ada before the process of selecting and configuring assets from the asset base is slated to begin. If necessary, the domain engineering team may demonstrate and explain the features of EDGE/Ada, in particular the Architecture Configuration Assistant (ACA), prior to entering the phase of application engineering in which the asset base services are directly needed. The designated application engineering team members will then use EDGE/Ada to make decisions about which assets to select and

how they are to be combined with each other. As a consequence of choices made by the application engineer, an internal data structure — the SUBSYSTEM EQUATION — will be created automatically. This data structure is then used to produce the Ada code that will be integrated with the rest of the code making up the final system application.

## Workproducts

■ SUBSYSTEM EQUATIONS

A SUBSYSTEM EQUATION is an internal data structure generated and updated by EDGE/Ada that documents the component choices made by the application engineer for each realm whose services are required in the final application. For any component that itself depends on realm parameters, the component selections made for each of these parameters is also defined by a clause within the set of SUBSYSTEM EQUATIONS. A printable form of the set of SUBSYSTEM EQUATIONS, using the syntax of the Architecture Specification, can be generated from EDGE/Ada. Exhibit 41 illustrates the appearance of a section of a SUBSYSTEM EQUATION listing for a subsystem to be derived from a preliminary version of the ELPA domain architecture.

## When to Start

- ASSET BASE APPLICATION PLAN sufficiently complete. The actual usage of the ASSET BASE, which is the main purpose of this task in the DAGAR process, should not be attempted before a sufficiently clear plan is in place for learning to what extent the functionality to be provided by the application is already available within the ASSET BASE. While alterations to the ASSET BASE APPLICATION PLAN can be expected to continue throughout any extended application development activity, the plan must be sufficiently clear at the beginning of the task so that engineers performing the task know what to expect from the ASSET BASE and realize what other resources besides those available from the ASSET BASE are required of the application.

- Capabilities available within the ASSET BASE needed to continue application development. Basically, serious investigation of the Asset Base and preliminary navigation through, and selection of, components from the asset base must take place as soon as the resources and services provided by the components are required to continue implementation of the application. Application engineers will turn to EDGE/Ada when that stage of the application development

```
system <Services> Test_Bed;

system <Fix_Calculation> Test_Fix;

system <Eigen_Calculation> Test_Eigen;

system <Coordinate_Transformations> Test_Transform

Test_Transform =
      Coordinate_Transformations_GRCS4 [Vectors_GRCS4,
Test_With_Files];

Test_Eigen = Eigen_Calculation_GRCS4 [Vectors_GRCS4,
      Test_With_Files];

Test_Fix = Fix_Calculation_EV[Vectors_GRCS4, Test_Transform,
      Test_Eigen, Test_With_Files];

Test_Bed = Test_ELPA[Test_Fix, Test_With_Files];
```

**Exhibit 41.** Sample set of Subsystem Equations

is reached that requires components from the asset base be made available for integration with the entire suite of application code.

## Inputs

- ASSET BASE. The ASSET BASE consists of the domain architecture in the form of an ARCHITECTURE SPECIFICATION along with a set of realms and components. The Architecture Configuration Assistant provides the application engineer with a guided tour through the ASSET BASE and enables choices to be made for each component slot contained in the architecture that is identified as being required in the application.

## Controls

- APPLICATION SYSTEM REQUIREMENTS. In learning about the ASSET BASE's features and characteristics, the application engineer is guided ultimately by the requirements at hand for the application. Along with the information about the assets themselves that is integrated into the ASSET BASE infrastructure and supporting materials, external circumstances and dependencies must be used to interpret this information and evaluate its importance and relevance.
- ASSET BASE APPLICATION PLAN. As mentioned several times previously in this document, this plan should be relied upon to give crucial guidance about how to approach the ASSET BASE and what needs the ASSET BASE can be expected to fulfill.

## Activities

The EDGE/Ada toolset provides the means for automatically generating a subsystem from the asset base, based on choices made by the application engineer. As introduced in the preceding section, the application engineer can use either the ACA or GLUE to pick components and compose the subsystem. Exhibit 42 shows a screen snapshot of the ACA being used to select a matrix component for use in an application subsystem. The application engineer can also obtain documentation and test materials for the chosen components from the asset base, if these are available.

The ACA is the preferred means of interacting with the asset base because the ACA automatically enforces component composition rules. It is impossible to produce a composition of assets from the asset base that breaks the component composition rules. Where there are unique component choices for a given realm, or where selection of a component in one part of the architecture forces selection of a unique component in another part, such selections are made automatically. As shown in Exhibit 42, selections made within the ACA are automatically reflected in the graphical architecture display provided through GLUE.

➤ Invoke ACA using the ASSET BASE data structures managed by EDGE/Ada

Using the consulting services of the domain engineering team as necessary for training and support, the application engineer begins by opening up an architecture view into the domain architecture through the ACA. The ACA will guide the user through a series of question and answer dialogues aimed at determining which component variants available at multiple points in the architecture are most appropriate to the needs and goals of the application engineer. Each answer given by the application engineer helps to narrow down the field of available choices and even make some consequential selections automatically. These selections are reflected in increasing amounts of detail being added to the SUBSYSTEM EQUATIONS being produced by the toolset. When the last question is answered, enough information has been obtained from the application engineer to completely specify which code specifications and bodies must be generated to give

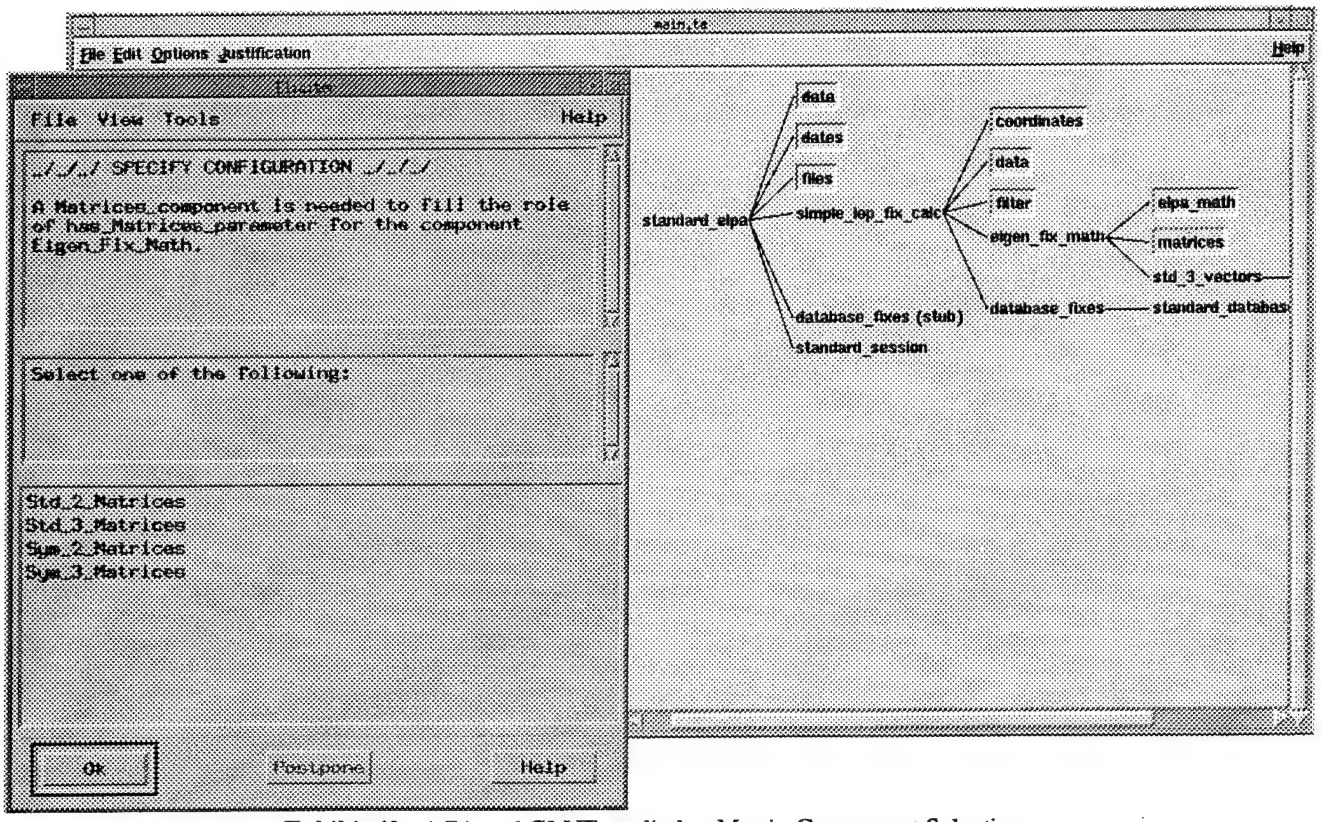



**Exhibit 42.** ACA and GLUE applied to Matrix Component Selection

the application engineer the best fit possible from the ASSET BASE. The graphical display is updated automatically as answers to questions lead to component selections being determined.

Alternatively:

➤ Invoke GLUE using ASSET BASE data structures.

If desired, the GLUE graphical display capability of EDGE/Ada can be accessed directly to see the alternatives that exist for each realm and for matching components within a realm for each of the realm parameters that exist for a particular component. These alternatives are displayed graphically as illustrated in the right hand side of Exhibit 42. There is no help offered in deciding which of the alternatives depicted in the graph are advantageous. The application engineer must be willing to examine Realm Descriptions, Component Specifications and Component Bodies to manually determine goodness of fit.

## When to Stop

- EDGA/Ada has produced a workable subset of SUBSYSTEM EQUATIONS. The stopping criteria for this task is easy to state: sufficient delineation of the available choices has led to a consistent and complete set of Subsystem Equations as produced by the appropriate EDGE/Ada tool. These equations must be complete enough to carry out the generation of Ada code for eventual integration into the overall application.

## Guidelines

- Application engineers require training to understand generator based approaches. Even good

Ada engineers need to be given time to learn and get comfortable with engineering techniques that incorporate fundamental dependence on the use of software generation. While learning how to use the toolset must be part of this training, this learning is not the most fundamental. A mindset is required that is open to code bodies and segments that are not handwritten but generated. A trust in the ability of the domain engineering team to produce efficient and correct components is also required

- Early, coordinated use of the ASSET BASE will help improve ASSET BASE quality. If possible, sustained and cooperative parallel activities coordinated across the domain and application engineering teams should be arranged to help lessen the risk of a mismatch between domain engineering activity output and required application engineering activity input.

# 7.3 Generate Subsystem Implementation

This task is a companion task to *Compose Subsystem* in that the subsystem configuration defined during the composition task must now be processed to produce the set of Ada files that implement the subsystem. These files are then integrated into the overall application by the application engineer. The system generation task is directly supported by EDGE/Ada where the act of generation is available as a command. The generated files are stored in the application engineer's working directory. If desired, Ada compilation of these files can be also be arranged from within the EDGE/Ada toolset.

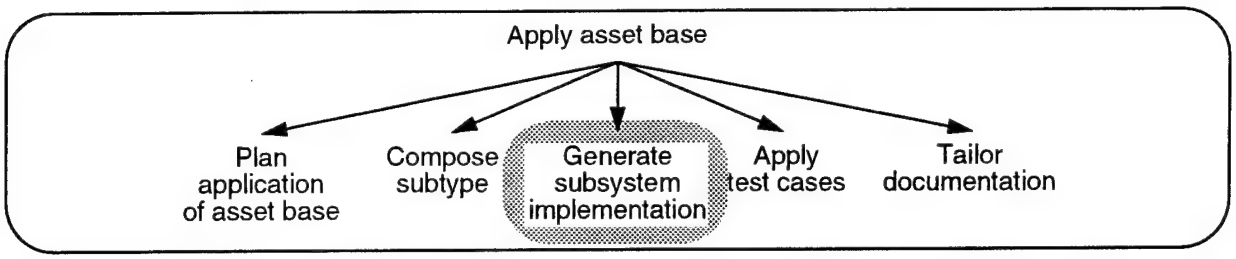



**Exhibit 43.** Generate Subsystem Implementation Process Tree

The primary purpose of the *Generate Subsystem Implementation* task is to prepare for final integration and use of the subsystem whose structure and components have been selected from the asset base. Rather than retrieving components from a reuse library, DAGAR components must be processed according to the realm parameter bindings that were made during subsystem composition. This processing will typically not require any manual intervention as the bindings are defined via the Subsystem Equations produced the ACA.

In the case that the "subsystem" is more or less stand-alone, the resulting Ada code will comprise a complete and testable system that is ready for routine evaluation and test procedures that will verify the suitability of the assets in their intended application context. However, in the general case, the generated subsystem forms a more complex portion of a larger system that must now be evaluated and tested in this larger context. Domain engineering, and in particular EDGE/Ada, cannot be expected to help the application engineer perform these integration activities. Depending on the extent to which manual steps must be taken to complete the integration, significant resources can be required to finally determine the quality and goodness-of-fit between the asset base and the needs of the current application.

## Approach

The standard approach to follow in generating the application subsystem is to use the services of EDGE/Ada which make the production of the subsystem a basically mechanical operation. For first-time uses of the ASSET BASE, and also for first-time or inexperienced users of the EDGE/Ada tools, the first subsystem generated may not have the best or expected properties when integrated into the rest of the subsystem. The application engineer may need to re-apply the *Compose Subsystem* task and following this, re-apply the *Generate Subsystem Implementation* task to better meet the needs of the application.

## Workproducts

- SUBSYSTEM

A DAGAR Subsystem is a set of Ada specification and body files that are able to be compiled and integrated (linked) with the rest of the application. All internal DAGAR dependencies and extended Ada statements have been replaced with compilable Ada that is ready to be compiled, linked and tested. Some examples of Ada code produced from DAGAR realm and component specifications are given in Exhibit 44 and Exhibit 45.

## When to Start

- SUBSYSTEM EQUATIONS complete. As soon as a complete set of bindings, normally through execution of the ACA, between components and associated realm parameters has been made, the application engineer is ready to begin subsystem generation. In the case of a large subsystem, with several separable branches, generation can begin for some of the branches while final composition and configuration activities are ongoing for the other branches.

## Inputs

- ASSET BASE. The Ada code to be produced in this task depends on the availability of a complete set of realms and components which are the core of the ASSET BASE.
- SUBSYSTEM EQUATION. As discussed above, the generation process relies on a collection of validated realm parameter to component bindings. These bindings are captured in a set of SUBSYSTEM EQUATIONS as illustrated in Exhibit 41.

## Controls

This task essentially has no controls of its own. Its performance is directly tied to the *Compose Subsystem* task and the controls of this task are indirectly applicable to the task of component generation.

## Activities

ACA and GLUE generate SUBSYSTEM EQUATIONS, based on the choices made by the application engineer. These equations contain complete information about the Ada code to be generated based on the application engineer's component selections. The EDGE/Ada toolset automatically generates Ada code for the subsystem based on the subsystem equations. If desired, the generated code can automatically be compiled through EDGE/Ada. The application engineer just needs to make the appropriate selections through EDGE/Ada's user interface.

For example, given the matrix realm and components shown in preceding exhibits (see Exhibit 24, "Partial Matrix Processing Services Realm Description," on page 46, Exhibit 32, "Complete Matrix Component Specification," on page 71 and Exhibit 34, "Partial Matrix Component Body," on page 76), the application engineer will have selected a particular component to instantiate the **`Matrix`** realm, and all of the components that must be matched against any realm parameters that the selected component requires. EDGE/Ada then produces the resulting Ada packages (specifications and bodies) that correspond to the architectural choices made. Exhibit 44 and Exhibit 45 illustrate segments of a generated Ada package specification and body when the **`Std_3_Matrices`** component is selected in the **`Matrix`** realm and when the **`Std_Vectors`** component is selected to match the **`Vector`** realm parameter.

```
-------------------------------------------------------------------
-- File: matrices.r
--
-- This is a realm  that defines operations between 2x2 symmetrical
matrices;
-- 3x3 symmetrical matrices; 2x2 non-symmetrical matrices;
-- 3x3 non-symmetrical matrices;
--
-------------------------------------------------------------------

with Std_3_Vectors;
with Computation_Support;
use Computation_Support;
with ELPA_Support; use ELPA_Support;
with Base_Support; use Base_Support;

-------------------------------------------------------------------
package std_3_matrices is

        type Matrix_Index is new Integer range 1..3;

        type Matrix is array
          (Matrix_Index, Matrix_Index) of Real;

function "+" (Left, Right : in Matrix)
      return Matrix;

function "*" (Left  : in Matrix;
              Right : in Std_3_Vectors.Vector)
      return Std_3_Vectors.Vector;

function "*" (Left  : in Std_3_Vectors.Vector;
              Right : in Matrix)
      return Std_3_Vectors.Vector;
-- ...
end std_3_matrices;
```

**Exhibit 44.** Generated Ada Matrix Package Spec

## When to Stop

- Compilable subsystem code available for integration. This task is a veritable push-button operation. It is over when the results of pushing the button have been generated and are ready for use in the rest of the application construction process. Later, upon attempting integration and subsequent application context testing, it may be determined that there are some problems with the code as generated. The best response in such a situation is to report problems back to the domain engineering team, or simply make alternate selections during a re-application of the *Compose Subsystem* task.

## Guidelines

- Attempt application integration early. If the ASSET BASE is constructed for early, prototypical use, the best usage approach will be, as early as possible during application engineering, to arrange for a trial application of the both the composition and generation tasks to produce actual code that can be integrated and tested in the expected application context. ASSETS in the ASSET BASE will have been tested in stand-alone fashion, but they cannot have been tested in many application contexts. Nor will the ease of use and integration of ASSETS produced from the ASSET BASE be assured, especially for early users of the asset base. As a risk

```
-------------------------------------------------------------------------------
--
-- File: std_3_matrices_b.k
--
-- This body implements the matrix operations for any 3*3 matrix
--
-------------------------------------------------------------------------------
--

package body std_3_matrices is

   ----------------------------------------------------------------------------
---

   function "+" (Left, Right : in Matrix) return Matrix is

      Result : Matrix;

-- ...
-------------------------------------------------------------------------------
-
   function "*" (Left  : in Matrix;
                 Right : in Std_3_Vectors.Vector) return Std_3_Vectors.Vector
is

      Vector_Index : Cartesian_Coordinates := Cartesian_Coordinates'First;
      Result       : Std_3_Vectors.Vector;

   begin -- "*"

      for Column_Index in Matrix_Index loop
         Result (Cartesian_Coordinates'Val (Column_Index - 1)) := 0.0;
         for Row_Index in Matrix_Index loop
            Result (Cartesian_Coordinates'Val (Column_Index - 1)) :=

               Result (Cartesian_Coordinates'Val (Column_Index - 1)) +
               (
                Left   (Column_Index, Row_Index) *
                Right  (Cartesian_Coordinates'Val (Row_Index - 1))
                );
         end loop;
      end loop;

  return Result;
```

**Exhibit 45.** Generated Ada Matrix Package Body

mitigation strategy, the application engineering team is encouraged to perform trial runs of component selection, generation and integration. Such trial runs will make members of the team more familiar with relevant pieces of the EDGE/Ada toolset and at the same time allow the team to experience integration and related testing experiences first hand. Depending on how robust and functional the selected, prototypical assets are, some preliminary integration testing can take place as well. If the asset base is complete enough, and the application contest is sufficiently stable that the nature of the services and coverage to be obtained through the asset base is known, candidate final asset base selections can be made. The application itself may still be skeletal in nature so in-context testing of the generated code may not be possible. However, early interface compatibility can be verified and if problems are discovered here, there will be sufficient time to correct such problems.

## 7.4 Apply Test Cases

As soon as the application context for which the ASSET BASE material created through the previous two tasks is sufficiently complete, the application engineer will typically apply test cases. These test cases, and the test procedures to apply them, may be made available through the ASSET BASE itself. However, because each application context is unique, the application engineer may need to modify the test material, the test procedures, or both to obtain adequate assurance that the code produced from the ASSET BASE is appropriate in the current context. In addition to test material provided through the ASSET BASE, test data and application-level test procedures available to the application engineering team through other sources may be applicable to the area of the application covered by the asset base. The ASSET BASE code must pass tests derived from this materials as well.

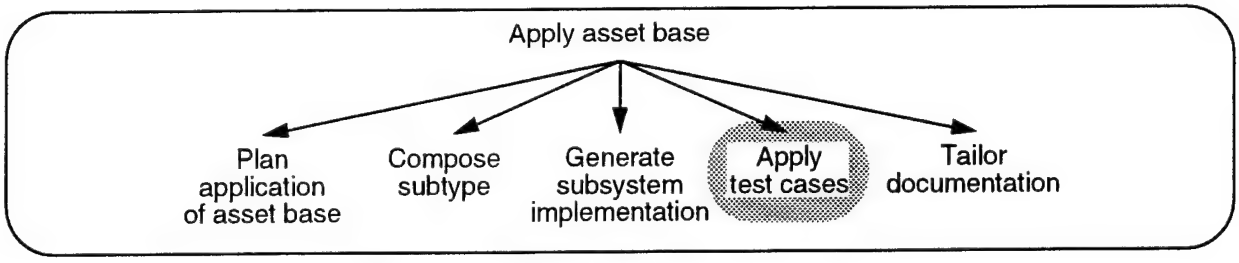



**Exhibit 46.** Apply Test Cases Process Tree

The primary purpose of the *Apply Test Cases* task is to verify the functionality and performance of those portions of the application produced directly from the ASSET BASE in the intended application context. Testing for ASSETS located in the ASSET BASE will have been done as part of domain engineering but such tests will have been, at best, applied in simulated application contexts. However, faults that are detected through application-specific testing may reveal internal faults in the ASSETS themselves that will need to be addressed by domain engineers maintaining the ASSET BASE.

In the heat of application development, defects in that portion of the application provided by ASSET BASE material may need to repaired manually within the code physically integrated into the application. There may be a tendency to ignore or postpone reporting such defects back to the appropriate domain engineering personnel. Such failures to report will eventually cause the ASSET BASE to deteriorate. Also, ASSET BASE defects reported by other application users that are eventually addressed by corrections to the Asset Base may then need to backfilled into application contexts that had not yet experienced the defect. If such fixes are merged into the application, an additional round of testing may be required to verify that no other defects are introduced.

### Approach

After composing and generating the ASSET BASE material in application-usable form, any available test material that is provided along with the ASSETS is assembled and prepared for use. Test procedures, test reporting practices and test application methods in effect within the overall application engineering effort are applied and test results recorded. Any subsystem software which fails to perform as expected must be analyzed further to try and learn the cause of the failure. Any failures traceable to the structure and content of code produced from the ASSET BASE must be reported back to the domain engineering team for corrective action. When necessary, local patches to integrated code may be necessary to keep projects on schedule. These patches should be considered only when waiting for official updates to the ASSET BASE results in unacceptable delays to application delivery schedules. Additional specialized testing (e.g. timing and capacity tests) may be required before the subsystem is determined to be ready for integration in a larger

system being developed by the application engineering project. There will be fewer defects for mature ASSET BASES with which there has been significant application engineering activity.

### Workproducts

■ TEST REPORTS

TEST REPORTS are those produced by the application engineering team for use within the application engineering effort per se. These reports are formatted according to application engineering standards and are combined with other application level quality assurance documents.

■ ASSET BASE TROUBLE REPORTS

ASSET BASE TROUBLE REPORTS are produced during application context testing but are meant for communication back to the domain engineering team. These reports address problems traceable to the form, behavior or results of application code that was derived from the ASSET BASE.

### When to Start

- Immediately upon integration of ASSET BASE material with application. It is never too soon to start testing activities. As long as a workable application framework is in place, and the code generated from the ASSET BASE is actually capable of producing results, these results should be checked against expectations. If the ASSETS themselves are only skeletal prototypes of what will later evolve into completely functional versions, early testing may be limited to checking interface correctness.

### Inputs

- ASSET BASE. In addition to the realms and components that make up the ASSET BASE, supplementary materials supporting use of the ASSETS will usually be available. Such testing materials can provide the basis for application-specific testing.

- SUBSYSTEM. The actual code produced from the ASSET BASE must be available in compilable form. Upon compilation and linking with the rest of the application, testing of the ASSETS in their intended application context can proceed.

### Controls

- ASSET BASE APPLICATION PLAN. Ad hoc testing should be avoided. A certain amount of testing of the ASSETS will have already occurred. Further testing should take place with a clear understanding of what is to be gained by the testing and what level of response there will be if system defects are detected as a result of the testing.

### Activities

A detailed description of testing activities is out of the scope of this document. The application engineering team is expected to have in place its own quality assurance process and the activities defined as part of this process will be followed here. The only aspect noteworthy of mention is that in deciding to use the results of domain engineering to partially fill needs within the application, the application engineering team has an obligation to report back testing results as these identify defects that are traceable to the ASSETS themselves. The ASSET BASE TROUBLE REPORTS

format should be followed in making these reports. Any regression testing activities that are part of the application testing process may need to be expanded to include the case where ASSET BASE updates may cause code that was generated from the asset base to become obsolete. Under certain circumstances, it may be necessary to decline the inclusion of newly generated code because the cost of re-validating the code as embedded in the application may be too high (assuming of course that the remedied defects do not directly impact the current application context).

### When to Stop

- All applicable test procedures have been applied. Testing can stop when none of the defects that the test data and test procedures were designed to expose are detected in the software under test. If test cases have led to the detection of defects, these defects are identified and appropriate modifications made. The tests are subsequently re-applied to see if the defects have been removed.

- No new ASSET BASE changes have been made. If there is parallel domain and application engineering activity, preliminary use of ASSETS from the ASSET BASE may need to be re-validated as newer versions of the Assets come available. At some point, ASSET updating will cease and the need to re-test code re-generated from the updated assets will disappear. For a rapidly evolving ASSET BASE with a tight application delivery schedule, the application engineering activity may need to halt its openness to new ASSET versions as long as the latest version integrated into the application passes the current test suite.

### Guidelines

- Begin testing activities as soon as possible. The earlier defects are uncovered, the easier they are to correct, and the more time exists in which to correct them. Preferably, Asset defects should be fixed by the domain engineering team members who are maintaining the Asset Base. If integration testing is scheduled early, there will be more time for this process to be completed.

- Assign specific application engineering team member to interface with Asset base maintenance staff. As discussed above, application engineering schedules may prevent timely dialogue between the application and domain engineering teams. Even if the application production schedule is not constrained, engineers often want to fix things the easy way, focusing on fixing things in place. Because the quality of the Asset Base depends on good and effective feedback, there is a danger that if problems go unreported, or not reported through proper channels, opportunities for improvements to the Asset base will be missed. To lessen the risk of this happening, a named advocate for cross-team communication should be appointed who has the responsibility of preparing the Asset Base Trouble Reports and coordinating Asset Base improvements with the application engineering team.

## 7.5 Tailor Documentation

In addition to ASSET BASE test materials that exist in template form and are then adapted for use in a particular application context, documentation material for ASSETS may also be available in the Asset Base. This material will also need to be tailored and completed for it to be useful in a particular application context. In the case of a large subsystem that at its lower levels can be completed in a variety of ways, documenting how the subsystem works completely will depend on which variants are in fact selected in a particular context. The upper levels of the documentation can have placeholders for describing what goes on at lower levels. But the substitution for these placeholders must take place after the component selections have been made. After an application-specific selection has been made, a full set of substitutions for placeholders within the various document templates can take place with a resulting complete set of documentation produced.

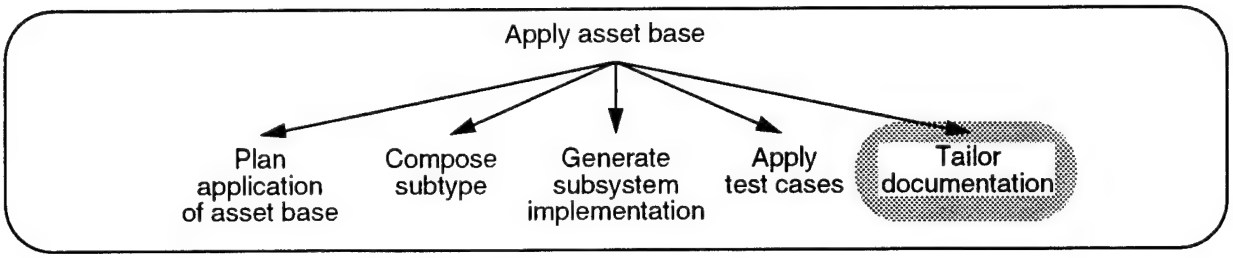



**Exhibit 47.** Tailor Documentation Process Tree

The primary purpose of the *Tailor Documentation* task is to make sure that after the determination has been made on how the ASSET BASE can be used to fill the requirements present in the application, the necessary descriptions of how these services work are obtained and any required user documentation that is derived from the Asset Base is integrated into the complete documentation set needed to support the application. While there is EDGE/Ada support for selection of ASSETS and generation of code based on these assets, there currently is no support for either application-specific testing or document production. Application engineers will normally have a number of document production tools at their disposal. The documentation material accessible through the ASSET BASE will need to be evaluated for applicability based on which components were selected for use in the application. Documentation related to these components will then be manually processed to extract sections to be integrated into the overall documentation set. It is expected that future versions of EDGE/Ada will support some form of document composition that corresponds to component composition.

Whereas it is important to test early and often, documentation preparation should be delayed until the selection of components to support code generation is finalized. As the final stages of generated code integration and testing is taking place, documentation specialists can weave into application system documentation sections appropriate to the components that have been finally selected. There is also a problem of updating system documentation in response to improvements in components and therefore component documentation. The component code generation update process is automated, but test and documentation updating is not. Needless re-work needs to be avoided, but improved components must eventually be reflected in improved documentation relating to those components.

### Approach

The documentation available from the ASSET BASE will typically be generic and will need to be tailored by the application engineer to use application-specific terminology and to meet system-specific documentation requirements. Application-level details are added to the documentation

and the documentation (including documentation for system users and maintainers) is transformed into its final form. If the documentation is constructed to make use of placeholders indicating where lower level details depend on choices available in the architecture, these placeholders will need to be replaced by detailed sections and passages that reflect which choice is made in the current application context.

### Workproducts

■ SUBSYSTEM DOCUMENTATION

The application engineer produces documentation using the tools and formats mandated by the application engineering team. Documentation sections that reflect ASSET BASE-supplied services must be integrated into the overall system documentation.

### When to Start

- Subsystem configuration derived from ASSET BASE complete. Because of the lack of documentation integration support, merging in documentation contributions from ASSET BASE materials should be delayed as much as the application engineering schedule permits.

### Inputs

- ASSET BASE. The ASSET BASE will normally contain documentation materials for most of the components available for selection from the asset base. These can be available as separate documentation files and may be written to reflect expected or possible composition relationships among the components in the ASSET BASE.

### Controls

- ASSET BASE APPLICATION PLAN. While the plan is an overarching control for all of the application engineering activities, this plan primarily acts as a control on the production of the final SUBSYSTEM EQUATIONS which control which components are used to generate the final subsystem code. These equations also determine any compositional relationships that can be applied to assembling document sections for integration into the application document set.

### Activities

➤ Review final SUBSYSTEM EQUATIONS.

As noted above, not only do these equations determine how application code is generated, they also indicate how structured documents that reflect the breakdown of functionality can be composed to tell the whole story using the detailed pieces that describe how individual lower level sections operate. For an ASSET BASE with a complete and segmented document set, where there is a one-for-one (or nearly so) correspondence between DAGAR components and document files (that describe the operations and services presented through these components), the SYSTEM EQUATIONS can be read to understand which document must get plugged into which other document to tell the big story in terms of the little stories.

➤ Integrate ASSET BASE document sections into Application documentation

This activity will be primarily a manual activity that converts the ASSET BASE document sections into the documentation format being used by the application engineering team. This activity will be harder if the services provided through the ASSET BASE are not well-localized in concentrated areas within the application system. If ASSET BASE documentation sections are themselves composed of other asset base sections, these local compositions should be accomplished first, again using the application engineering team's documentation tools. In the course of performing this integration, any application-specific editing of the wording used in the ASSET BASE sections can be carried out.

## When to Stop

- ASSET BASE documentation integrated in Application documentation set. This documentation production activity will often be one of the last activities of a combined domain engineering/application engineering activity. The task is over if the application's document set fully describes what the application does, and if the sections derived from the ASSET BASE have been fleshed out and are well blended-in.

## Guidelines

- Delay documentation related to ASSET BASE as much as possible. The reasons for this guideline have already been given. There will be less of a reason to delay once some EDGE/Ada support exists for document composition to parallel the support already in place for code component composition.
- Pick ASSET BASE document tools and formats in common application usage. Because the documentation process is still unautomated, the less document conversion/re-formatting that is required, the better. The domain engineering team should identify and adopt those documentation formats that are already being used in applications that are potential ASSET BASE customers. If multiple formats are in use, both application and engineering teams should invest in some document conversion infrastructure to easily convert content in one format to another.

# Part III: Applying DAGAR

Part II presented a detailed description of the DAGAR process model. The model offers considerable guidance for performing the DAGAR activities:

- Defining an asset base architecture, based on a prescriptive domain model that describes the requirements for the asset base architecture
- Implementing assets within the framework of the domain architecture
- Application of the domain assets through the selection of assets for a particular application, or part of an application.

The purpose of Part III is to offer some general guidance in how to apply DAGAR with specific domain engineering methods.

- Section 8 provides a description of how DAGAR can be used to support definition of an asset base architecture and implementation of the asset base within the ODM domain engineering life cycle. The section describes how DAGAR fits into ODM and how ODM can be tailored to provide more direct support for the development of a DAGAR asset base architecture.

# 8.0 DAGAR as a Supporting Method of ODM

Although DAGAR can be used with other domain engineering methods, DAGAR directly supports Asset Base Architecture Definition and Asset Base Implementation as part of the ODM domain engineering life cycle. Section 8.1 gives an overview of the ODM domain engineering life cycle. Section 8.2 explains how DAGAR can fit into the overall ODM life cycle is described. Section 8.3 gives some guidelines on tailoring ODM when used with DAGAR.

## 8.1 The ODM Domain Engineering Life Cycle

ODM domain engineering begins by planning the domain to be engineered. Project objectives are set in relation to project stakeholders. Candidate domains are characterized and the domain of focus is chosen. The selected domain is then defined by formally stating what is entailed by the domain and clarifying what is in and out of the domain. The domain definition includes a *domain interconnection model*, which defines the relationship between the domain of focus and other domains.

Once domain planning has been completed, the domain is modeled. Domain modeling includes developing descriptive models that document what has been implemented in the past and extending these models to include what might be implemented in the future. The *domain model* produced describes common and variant *features* within the domain and rationale for the variations. The domain model will be used as a basis for selecting the range of variability to be supported by assets in the asset base. A secondary product of the domain modeling phase is a *domain dossier*, which documents the specific information sources used as a basis for modeling. The domain dossier will be used to trace legacy artifacts that are candidates for reengineering into reusable assets and to identify constraints in systems that assets will be migrated into.

After domain modeling, the asset base is scoped to derive an overall feature profile for the asset base. Asset base scoping also includes characterizing the market for the asset base, that set of application system contexts in which practitioners will potentially utilize domain assets. The key function of this activity is to derive a subset of the features and potential customer contexts described in the domain model that will be supported by the asset base. The resulting product is called the *asset base model*. The asset base includes a map between customers and features showing potential customers for features. A secondary product of asset base scoping is the *asset base dossier*. The asset base dossier includes the domain dossier developed during domain modeling and information about customer contexts for the customers that will be supported by the asset base.

Once the asset base has been scoped, an asset base architecture and assets are developed. Architectural concerns include ascertaining to what extent the design and implementation of assets is constrained by the external interfaces anticipated by different customer contexts, as well as determining the internal structure and interconnections of components in the asset base. An architecture is defined that adheres to these constraints.Then assets are implemented for portions of the asset base. These assets can be implemented either using a generative or a constructive technique.

## 8.2 How DAGAR supports ODM

As shown in Exhibit 48, the *Define Asset Base Architecture* and *Implement Asset Base* phases of DAGAR can be used to perform the *Define Asset Base Architecture* task, and *Implement Asset Base* sub-phase of ODM. Assets are implemented in DAGAR using a generative approach, so the ODM *Implement Asset Base* sub-phase is tailored in DAGAR to follow a generative asset approach. Also, much of the infrastructure for applying the asset base is already implemented in

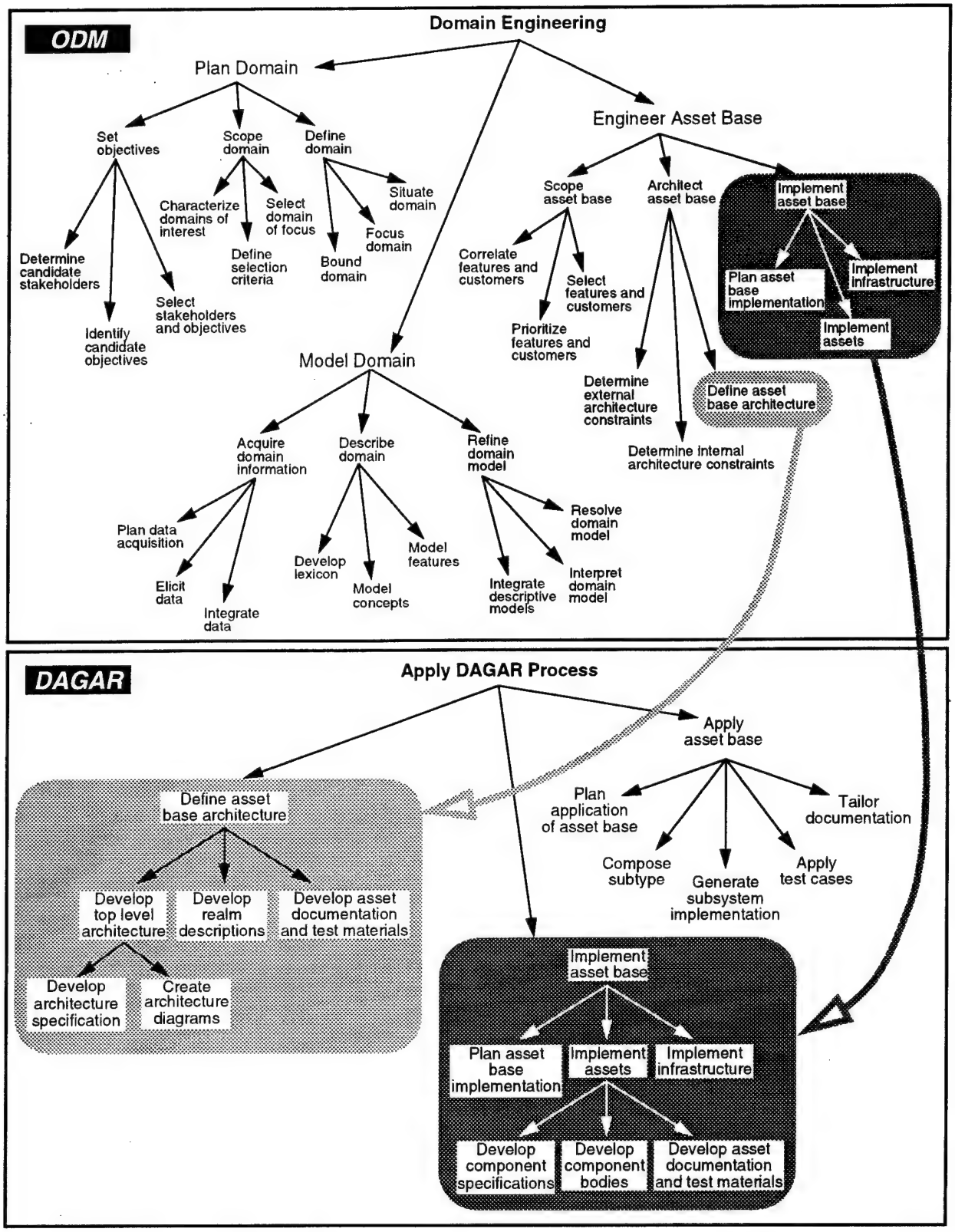



**Exhibit 48.** DAGAR as a supporting method of ODM

the EDGE/Ada toolset, so the DAGAR *Implement Infrastructure* task is confined to the determination of asset constraints. Since ODM is a Domain Engineering method which does not include how to apply the asset base, the *Apply Asset Base* phase of DAGAR is not covered in ODM.

## 8.3 Tailoring ODM for use with DAGAR

As discussed in [17], ODM can be tailored for an easier transition from an ODM Asset Base Model to a DAGAR Asset Base Architecture. For transition to DAGAR, ODM model development activities should be more focused with an early and continuing emphasis on the utility and appropriateness of model information being filtered and interpreted from an architecture-centric viewpoint. it is important that the description of features in the asset base model be approached from an architectural perspective and that the elements that are included in the asset base dossier include sufficient content to be able to determine the architectural implications of feature combinations. There must have been a significant study made of the domain (including system that include the domain), and a sufficient amount of information produced and organized into models, that the architecture can be defined in terms of the models, and in terms of the expected and planned customer needs for systems that will be created from the asset base. It should not be necessary to return to the exemplars or customers to determine relevance or make architectural decisions.

Below are some further lessons learned about tailoring ODM for use with DAGAR:

- **Establish domain architecture objectives and techniques early in the domain planning process.**

  Product line development should take place with a clear understanding of what architectural methods are to be employed. If it is necessary to delay final commitment to an architectural method, at the very least domain model development staff should be aware of the alternatives being considered. While ODM itself is not oriented toward a particular architectural method, model development activity should take place with an awareness of what architectural method is planned or expected to be used.

- **Keep in mind long-term goals when producing ODM descriptive domain models.**

  While it is important to gather sufficient domain information so that the breadth and depth of domain functionality and requirements are understood, the development of these preliminary models should not be viewed as an end in themselves. They should serve the goal of producing focused and effective descriptive feature models. In fact, it may be preferable to make production of the three descriptive models selected by the demonstration project (lexicon, stakeholder and descriptive feature model) the primary emphasis of the domain modeling effort. A domain concept model would only be developed as necessary to support the structure and content of the feature model. The role that the concept model serves in the ODM process may be better served by an expanded Lexicon model.

- **Choose appropriate model representations for each of the model types — different models may be better understood and organized using simpler representations.**

  The demonstration project primarily used the RLF mechanism to store and organize domain information. In particular, the domain lexicon model, the descriptive (and later the prescriptive) feature model and the domain stakeholder model were all developed as RLF models. For certain kinds of information, using a semantic formalism provided by a tool such as RLF is an unnecessary complication that can delay the completion of these models. For example, the lexicon, as a highly changeable list of terms and definitions, preferably linked so that related terms can be accessed easily, is more naturally represented as a hypertext structure

such as can be provided using HTML. With the most recent enhancement of the RLF to become OpenRLF, with a complete Web browser interface and easy connectivity to other kinds of web-based data, the lexicon could be managed as pure HTML text with appropriate links from and to RLF models.

- **Look for, and carefully analyze, exemplar architectural material.**

  Older systems are not likely to come equipped with system architecture diagrams or CASE tool databases where exemplar architectural information is likely to be found. However, it should be possible to examine existing artifacts to locate and extract information about how the system is put together and how its de facto package of services is made available to users of the system. Note that users may be human operators or other tools or other system components. Where the information may not exist directly, it may be available in the heads of system developers and while it may take some doing to track down and interview development staff, the payoff for such an effort can be enormous. While it will always take a significant effort to look at old code or documentation, if care is taken not to drown in a sea of detail, a controlled amount of reverse engineering can lead to useful system level views of the exemplar architectures. In comparing these views to one another, valuable insight into domain commonality and variability will be obtained. Such insight will inform and guide the process of creating the domain architecture, at least in its early phases.

- **Consider the use of SAAM (Software Architecture Analysis Method) diagrams as a means of finding a common representation for multiple system architectures.**

  SAAM [8] describes work at the SEI aimed at looking at families of related systems (domain exemplars in ODM terms) and providing pictorial representations of the large-scale architectural elements within these systems. Once the exemplar system structures have been expressed using a particular notation, it is significantly easier to compare and contrast them and thereby see what has "worked" architecturally in the past within the chosen domain. It may be possible to use the SAAM diagrammatic formalism to derive a generic system architecture that includes those common service groups and boundaries that were seen during analysis of the exemplar systems. Where significant variability exists, the generic architecture can leave the corresponding architecture segment as undefined. While the RLF is not useful for drawing such SAAM diagrams, it is useful for modeling the interconnection between, and providing access to, a collection of such diagrams for various systems and system views. Through the RLF action mechanism, the diagrams themselves may be viewed by invoking the corresponding viewing program.

- **While draft versions of the Domain Lexicon and Stakeholder models may be the best early models to produce, it is never too soon to think about feature models during the early weeks of domain analysis and modeling.**

  By concentrating on identifying interesting architecture structure and communication methods early in domain modeling, it will be less likely that domain modeling will be pursued for its own sake wherein interesting domain information tidbits are identified and organized into model structure. Such activity is understandable if domain education is a principal purpose for doing domain modeling. If the principal purpose is the production of a product line architecture, every effort should be taken to minimize the amount of peripheral information that is examined and modeled. By being feature conscious from the very beginning, extraneous information will be easier to recognize and skip over.

- **By performing the tasks of domain analysis and modeling, project staff will over time become reliable domain informants themselves especially in regard to architectural concerns.**

  In doing the comparative analysis that is the hallmark of domain engineering methodology, domain engineers will become sufficiently expert that their views of desirable architectural organization and interconnection may point the way to possible product line architecture decisions and configurations. Some of these insights can be quite novel in comparison with existing systems. In fact, innovation modeling is one cornerstone of ODM. During domain engineering, staff must be open to the occurrence of these insights. Certainly all such insights or novel ideas should be recorded for later careful deliberation and possible verification of workability or utility by an external domain expert.

- **Spend little time reviewing exemplar source material that is light on structural or feature combination information.**

  In rapidly moving from domain models to domain architecture, it will be necessary to quickly review and categorize a large amount of data obtained from exemplar information sources such as user manuals, requirements documents, design documents, expert interviews and of course exemplar system code. While every source document will probably contain some useful bit of information, many of them will be light on architecturally relevant information: how things are put together, how system elements communicate with one another, how feature/functional elements are bundled together or co-located within the same or affiliated sections of system code, what features/services are actually available across the system element interfaces, etc. With experience, the domain engineer will be able to quickly tell what information sources are likely repositories of useful information and which ones merely repeat what is already known or contain non-useful sorts of information.

- **Complete an early version of the prescriptive feature model and determine whether the resulting model structure appears suitable for use in deriving a product line architecture.**

  While there is a natural transition from descriptive modeling to prescriptive modeling, it is not necessary to wait until the period allocated to descriptive feature modeling has been completed. The events outlined in the ODM process are not meant to be viewed or executed sequentially and in a shortened and focused domain modeling activity, some degree of parallelism is a necessary consequence. If the descriptive feature model presents an "as-is" view of the space of domain features (including innovative derivations of a set of "what-if" features), the prescriptive feature model should present the set of "to-be" features for the product line. It will help the process of determining the planned feature set if several drafts of the set are produced at suitable intervals within the scheduled model development period. Continuing with this suggestion of early model refinement, an effort to develop an early draft DAGAR domain architecture to which elements of the prescriptive feature have been mapped is also advisable.

# Appendix A: DAGAR Process Model

This appendix presents the DAGAR process modeled using the Process Tree and $IDEF_0$ notations. The full Process Tree diagram and the set of $IDEF_0$ diagrams appear on the pages that follow. The order in which the $IDEF_0$ diagrams appear is consistent with the hierarchical process decomposition structure of DAGAR and should be straightforward to follow.

Organizations can use the Process Tree and $IDEF_0$ model directly as a basis for modeling DAGAR in more detail. They can also adapt the processes and information availability to address their specific needs, or they can integrate the model (or some adaptation thereof) with existing $IDEF_0$ process models.

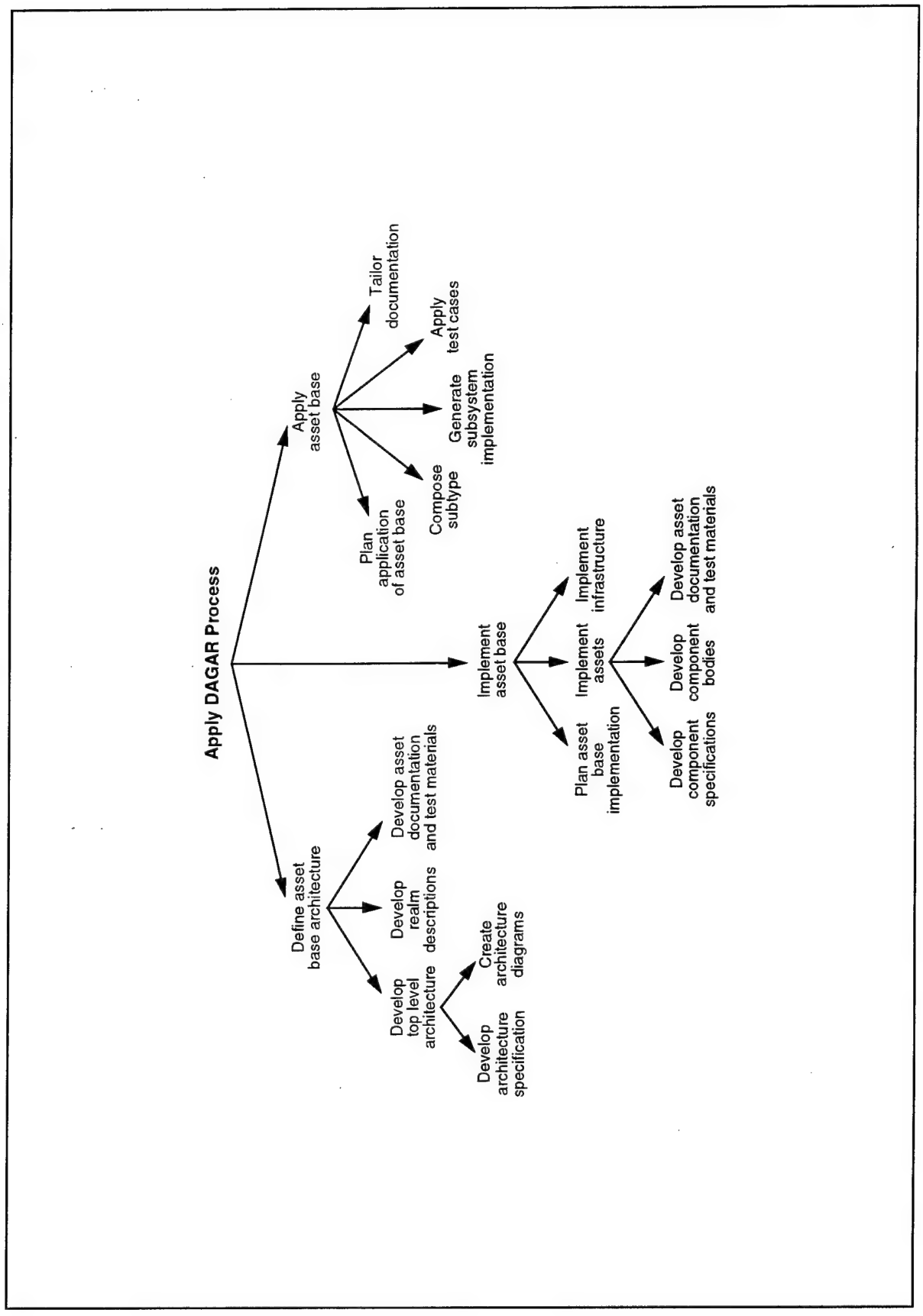
Process Trees were developed on the Army/Lockheed Martin Tactical Defense Systems STARS Demonstration Project to depict multiple levels of functional decomposition of a process in a hierarchical, graphical representation. A Process Tree decomposition appears as activities connected by arrows. Arrows decompose an activity to its subactivities. Process Trees provide a convenient way of navigating complex process decompositions. Nodes within Process Trees are used to index into the DAGAR $IDEF_0$ Diagrams.

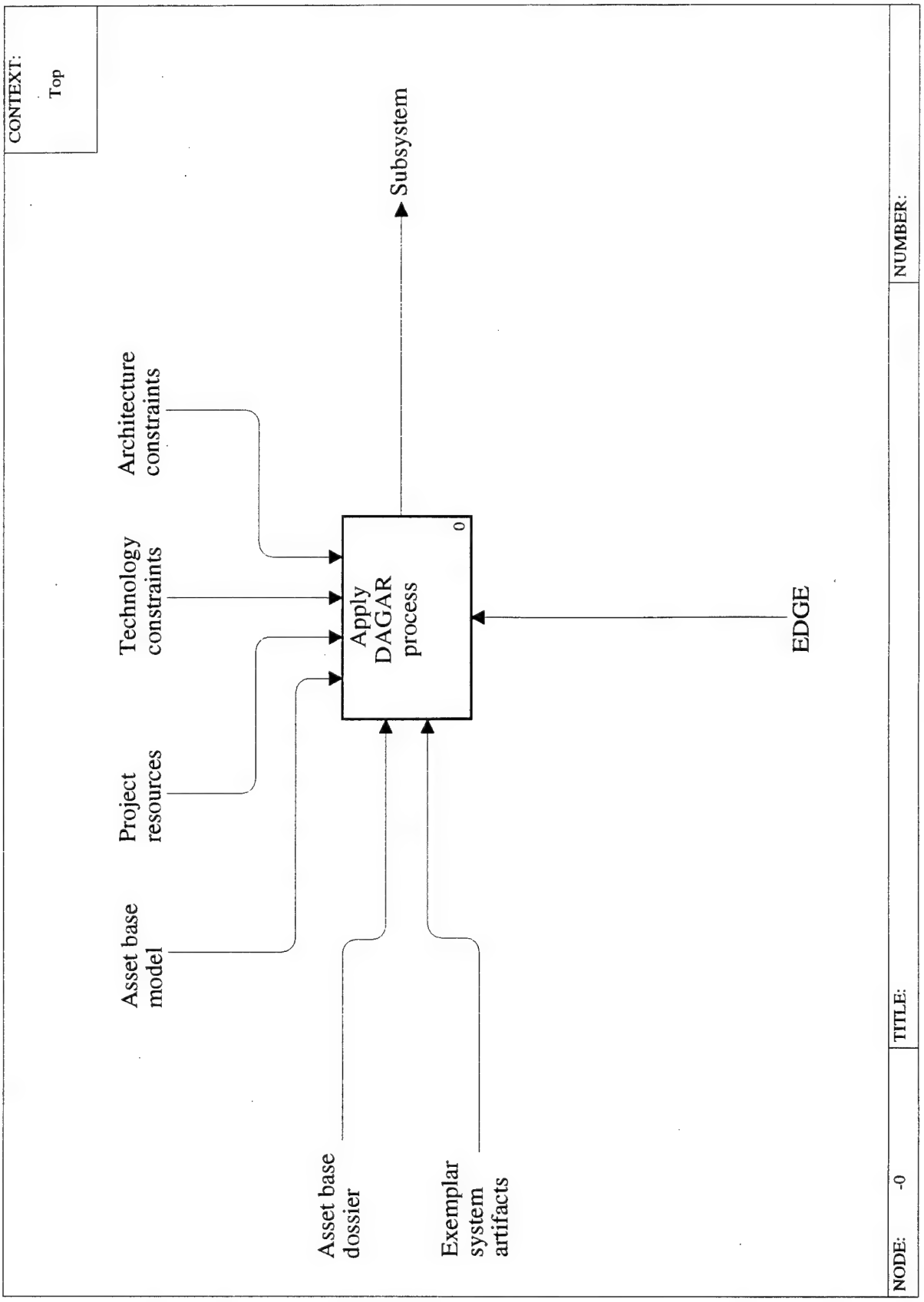
$IDEF_0$ is becoming an increasingly popular process modeling notation, but not all readers of this document may be familiar with it. $IDEF_0$ is based on the Structured Analysis and Design Technique (SADT) — a graphical approach to system description.

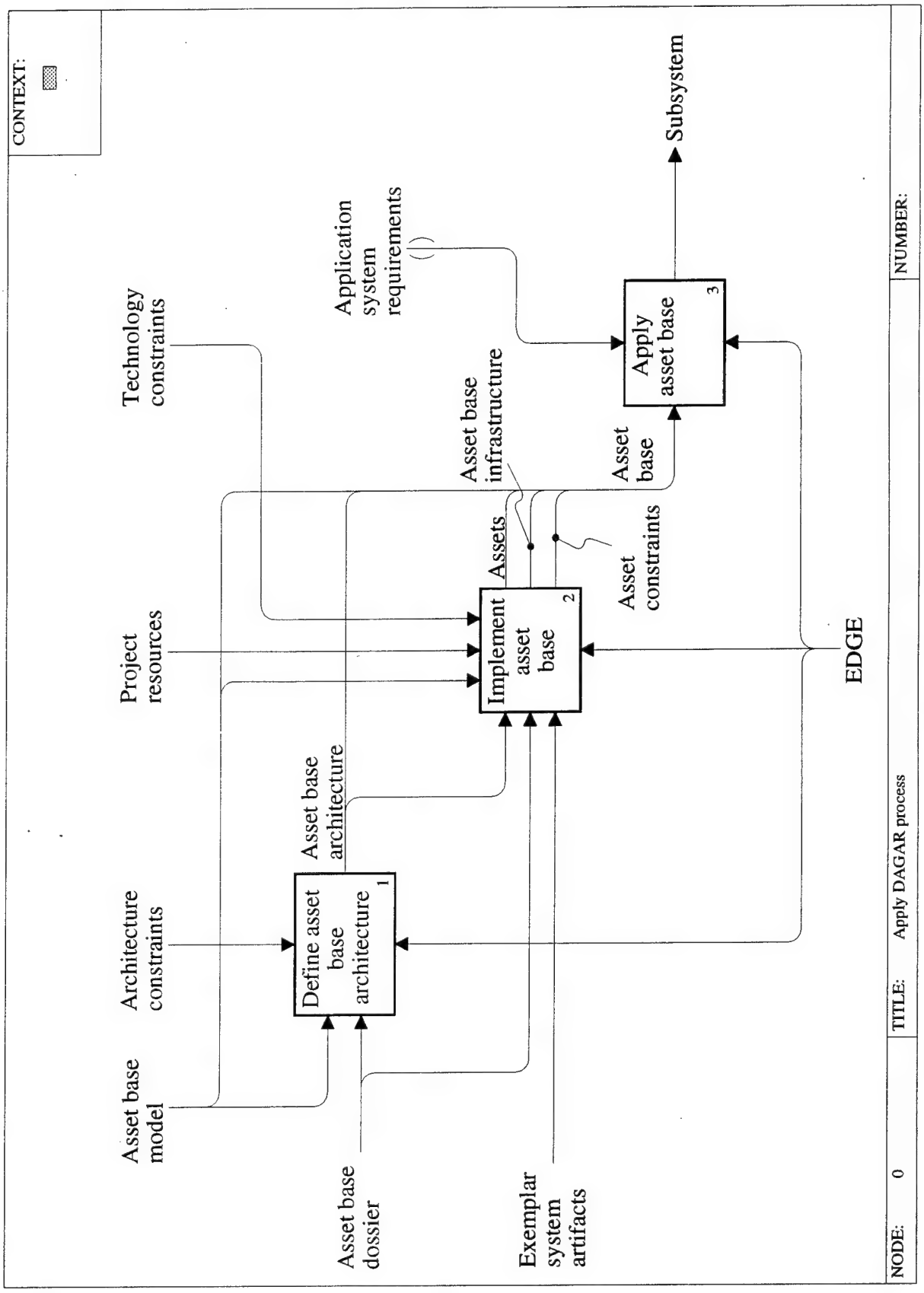
An $IDEF_0$ Activity Diagram contains one level of decomposition of a process. Boxes within the diagram depict the subactivities of the parent activity named by the diagram. Arrows between the boxes depict availability of work products to activities. Arrows entering the left side of a box are inputs to an activity. Arrows exiting the right side of a box are outputs from an activity. Arrows entering the top of a box are controls that regulate the activity. Arrows entering the bottom of a box are mechanisms that support the activity.

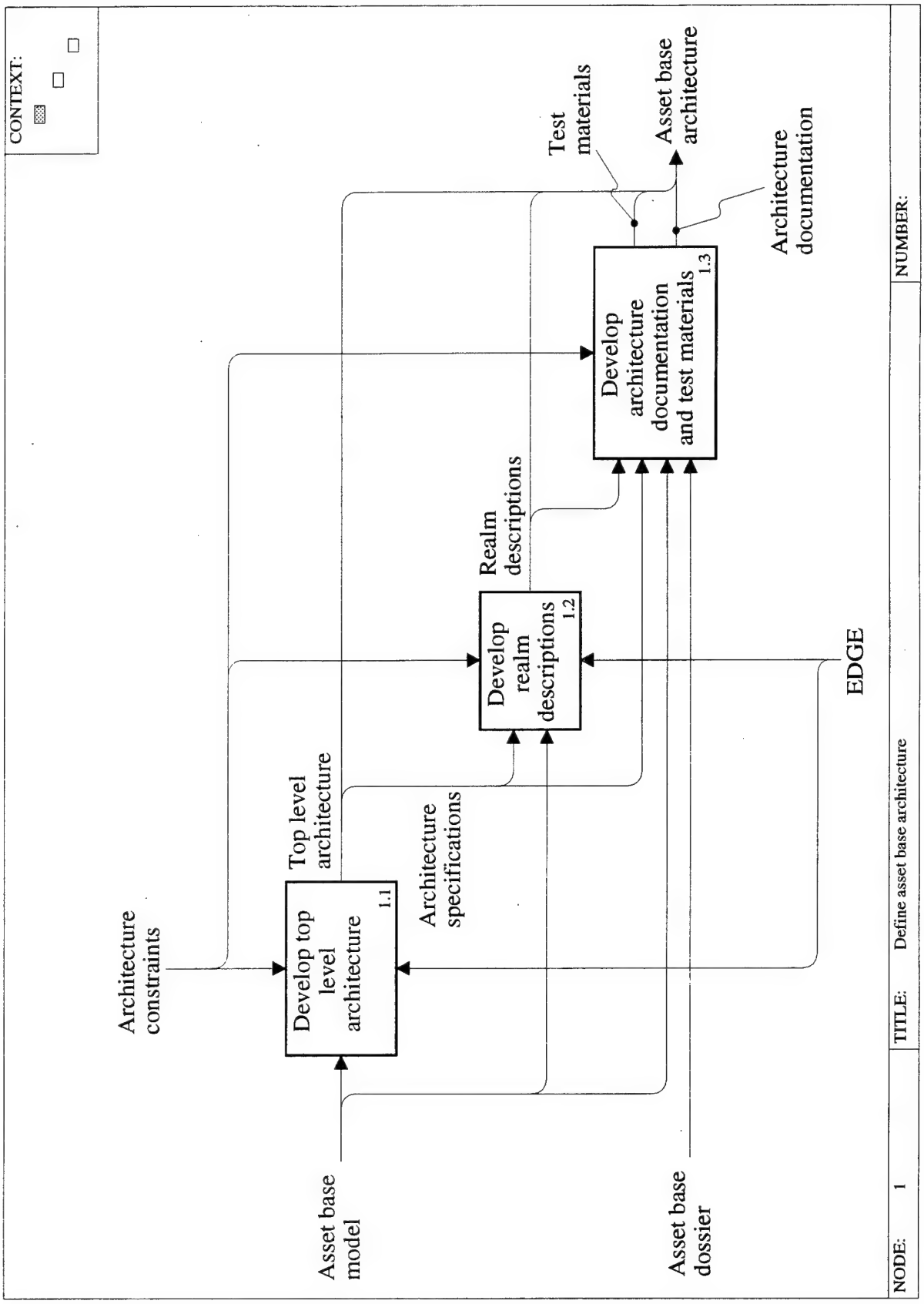
Note that a sequential ordering of boxes in the diagram does **not** imply a sequential flow of control between the activities. In many cases activities can be performed in parallel. Often feedback from a latter activity will result in returning to a previous activity and revising the work products produced.

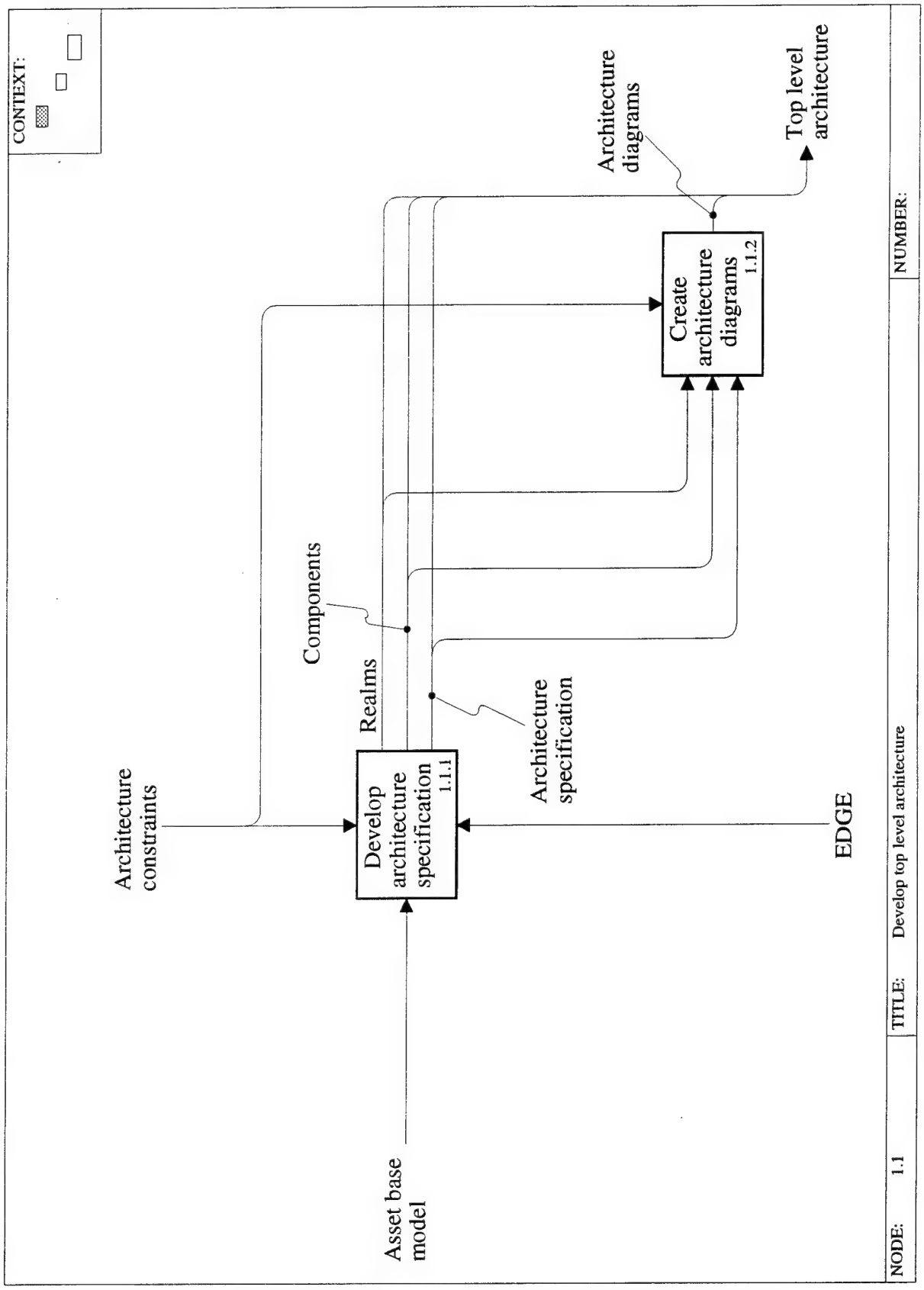
See [10] and [14] for more details on $IDEF_0$ syntax and semantics.

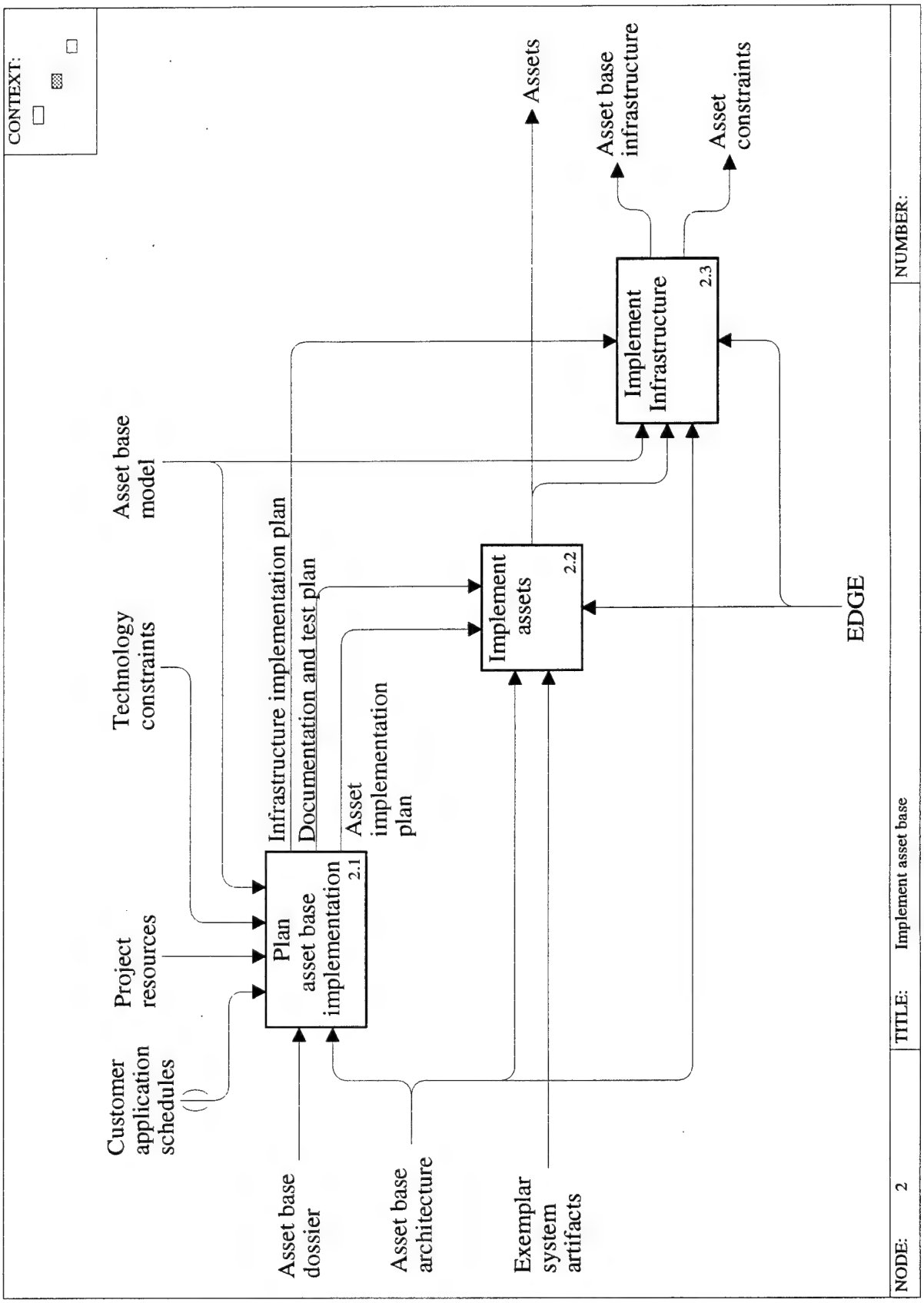

Apply DAGAR Process
Define asset base architecture
Develop top level architecture
Develop realm descriptions
Develop asset documentation and test materials
Develop architecture specification
Create architecture diagrams
Apply asset base
Plan application of asset base
Compose subtype
Generate subsystem implementation
Apply test cases
Tailor documentation
Implement asset base
Plan asset base implementation
Implement assets
Develop component specifications
Develop component bodies
Implement infrastructure
Develop asset documentation and test materials

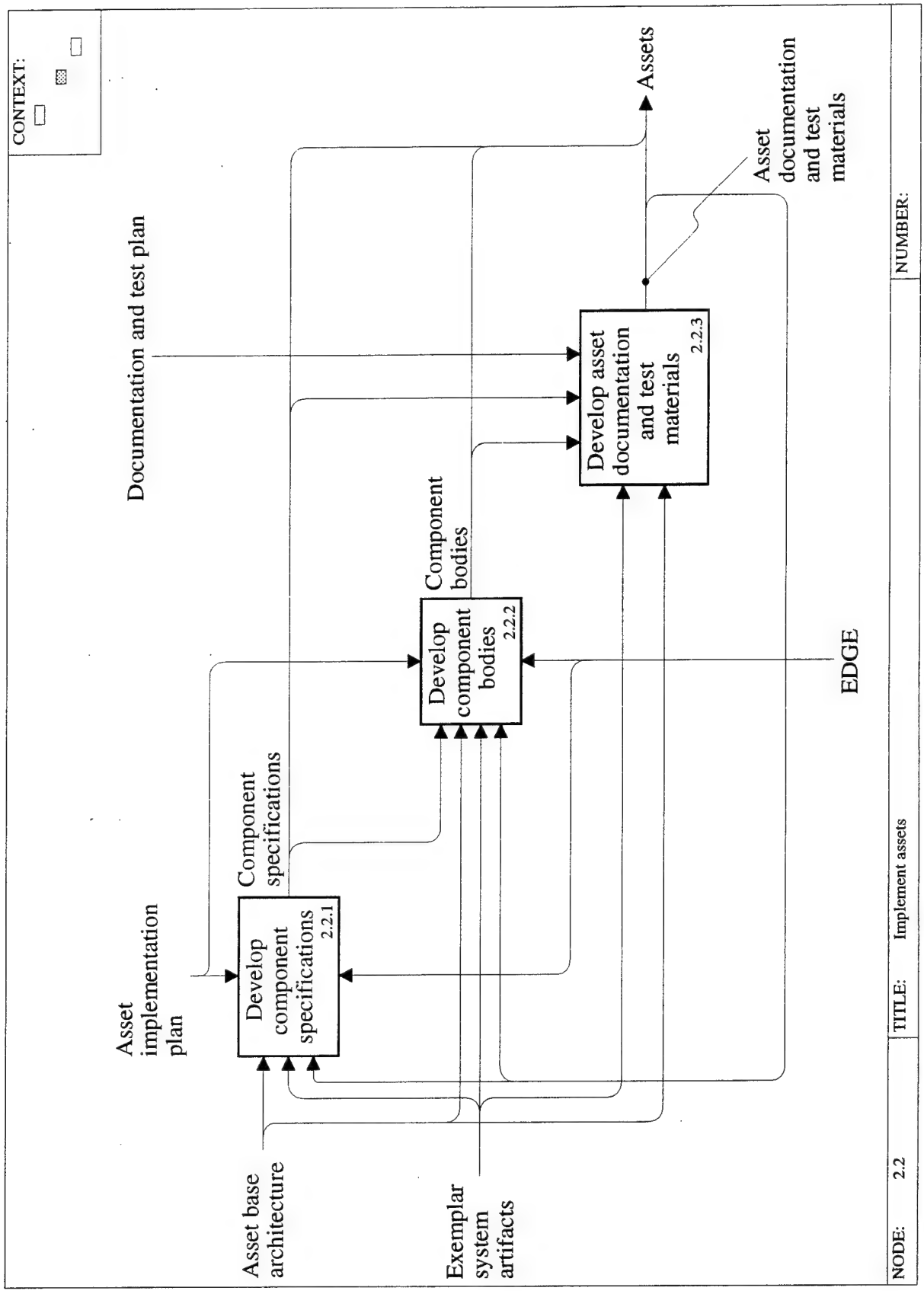

CONTEXT:
Top
Subsystem
Architecture constraints
Technology constraints
Project resources
Asset base model
Apply DAGAR process
0
EDGE
Asset base dossier
Exemplar system artifacts
NODE: -0
TITLE:
NUMBER:

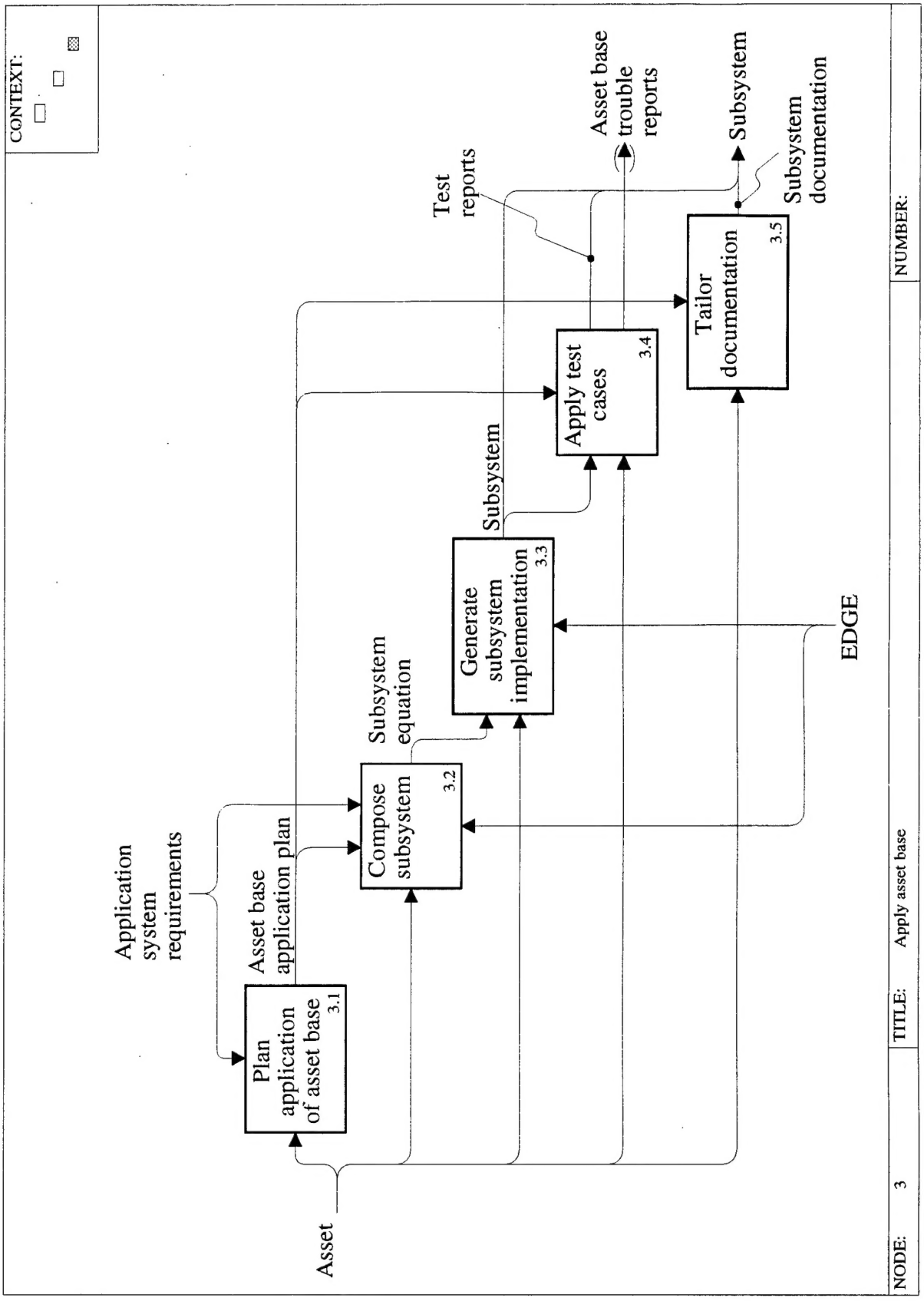

CONTEXT:
Technology constraints
Application system requirements
Subsystem
Asset base infrastructure
Asset base
Apply asset base
3
Assets
Asset constraints
Project resources
Implement asset base
2
EDGE
Asset base architecture
Architecture constraints
Define asset base architecture
1
Asset base model
Asset base dossier
Exemplar system artifacts
NODE: 0
TITLE: Apply DAGAR process
NUMBER:


CONTEXT:
Architecture constraints
Develop top level architecture
1.1
Top level architecture
Architecture specifications
Develop realm descriptions
1.2
Realm descriptions
Develop architecture documentation and test materials
1.3
Test materials
Asset base architecture
Architecture documentation
EDGE
Asset base model
Asset base dossier
NODE: 1
TITLE: Define asset base architecture
NUMBER:


CONTEXT:
Architecture diagrams
Top level architecture
Create architecture diagrams
1.1.2
Components
Realms
Architecture specification
Architecture constraints
Develop architecture specification
1.1.1
EDGE
Asset base model
NODE: 1.1
TITLE: Develop top level architecture
NUMBER:


CONTEXT:
Customer application schedules
Project resources
Technology constraints
Asset base model
Asset base dossier
Asset base architecture
Exemplar system artifacts
Plan asset base implementation
2.1
Infrastructure implementation plan
Documentation and test plan
Asset implementation plan
Implement assets
2.2
Implement Infrastructure
2.3
Assets
Asset base infrastructure
Asset constraints
EDGE
NODE: 2
TITLE: Implement asset base
NUMBER:


CONTEXT:
Documentation and test plan
Asset implementation plan
Develop component specifications 2.2.1
Component specifications
Develop component bodies 2.2.2
Component bodies
Develop asset documentation and test materials 2.2.3
Assets
Asset documentation and test materials
EDGE
Asset base architecture
Exemplar system artifacts
NODE: 2.2
TITLE: Implement assets
NUMBER:


CONTEXT:
NODE: 3
TITLE: Apply asset base
NUMBER:
Application system requirements
Asset base application plan
Plan application of asset base 3.1
Compose subsystem 3.2
Subsystem equation
Generate subsystem implementation 3.3
Subsystem
Apply test cases 3.4
Test reports
Asset base trouble reports
Tailor documentation 3.5
Subsystem
Subsystem documentation
EDGE
Asset

# References[1]

[1] Batory, Don., "le: A Type Expression Language" Loral Federal Systems/University of Texas Tech. Report ADAGE-UT-93-02. May 1993.

[2] Batory, Don., "Software System Generators, Architectures and Reuse" Tutorial Presented At The 3rd International Conference On Software Reuse, Rio de Janiero, Brazil, 2 November 1994.

[3] Batory, D., Coglianesi, L., Goodwin, M., and Shafer, S. "Creating Reference Architectures: An Example from Avionics," *Proceedings of ACM-SIGSOFT Symposium on Software Reusability (SSR'95)*, Seattle, WA, April 1995.

[4] Batory D., and O'Malley, S. "The Design and Implementation of Hierarchical Software Systems with Reusable Components." *ACM Trans. Software Eng. and Methodology*. October 1992. pp. 355-398.

[5] Batory, D., Singhal, V., Thomas, J., Dasari, S., Geraci, B., Sirkin, M."The GenVoca Model of Software-System Generators," *IEEE Software*, September 1994, pp 89-94.

[6] Goguen, J. "Reusing and Interconnecting Software Components." *IEEE Computer*. February 1986.

[7] Habermann, A. "Modularization and Hierarchy in a Family of Operating Systems." Carnegie Mellon University Tech. Report CS-78-101. February 1978.

[8] Kazman, Rick, et al., "SAAM: A Method for Analyzing the Properties of Software Architectures." *Proceedings of the Sixteenth International Conference on Software Engineering*, May 1994.

[9] Klingler, Carol D., and Schwarting, Dan, "A Practical Approach to Process Definition," *Proceedings of the Seventh Annual Software Technology Conference*, Salt Lake City, Utah, April 1995.

[10] Marca, David A., and McGowan, Clement L., *SADT, Structured Analysis and Design Technique*. McGraw-Hill, New York, NY, 1988.

[11] Moore, G. *Crossing the Chasm: Marketing and Selling Technology Products to Mainstream Customers*. Harper Business, New York NY, 1991.

[12] Parnas, D. L. "On the Design and Development of Program Families." *IEEE Trans. Software Engineering*. March 1976.

[13] Sirkin, M. *A Software System Generator for Data Structures*. Doctoral Dissertation. University of Washington, Seattle. 1994.

[14] Softech Inc., Integrated Computer-Aided Manufacturing (ICAM) Architecture Part II. Volume IV - Function Modeling Manual ($IDEF_0$). Technical Report AFWAL-TR-81-4023 Volume IV, Materials Laboratory (AFWAL/MLTC), AF Wright Aeronautical Laboratories (AFSC), Wright-Paterson AFB, Dayton, Ohio, June 1981.

1. STARS documents can be obtained electronically from ASSET and in hard copy from DTIC.

[15] Software Technology for Adaptable, Reliable Systems (STARS), Architecture Configuration Assistant Users Manual. Technical Report STARS-PA19-S001/003/00, January 1996.

[16] Software Technology for Adaptable, Reliable Systems (STARS), Army STARS Demonstration Project Experience Report. Technical Report STARS-AC-A011R/003/02, April 1996.

[17] Software Technology for Adaptable, Reliable Systems (STARS), Bridging the Gap Between Domain Modeling and Domain Architecture Definition. Technical Report STARS-PA19-S004/001/00, March 1996

[18] Software Technology for Adaptable, Reliable Systems (STARS), EDGE Users Manual, Version 2.1. Technical Report STARS-PA19-S005R1/002/00, April 1996.

[19] Software Technology for Adaptable, Reliable Systems (STARS), Enhanced Domain Generation Environment (EDGE) Source Code Release Version 2.1, SunOS Implementation. Version Description Document STARS-PA19-S006R1/001/00, April 1996.

[20] Software Technology for Adaptable, Reliable Systems (STARS), Organization Domain Modeling (ODM) Guidebook, Version 1.0. Technical Report STARS-VC-A023/011/00, March 1995.

[21] Software Technology for Adaptable, Reliable Systems (STARS), Reuse Library Framework (RLF), Version 4.2, User's Manual. Technical Report, June 1995.

[22] Software Technology for Adaptable, Reliable Systems (STARS), STARS Conceptual Framework for Reuse Processes (CFRP), Vulume I: Definition, Version 3.0. Technical Report STARS-VC-A018/001/00, October 1993.

[23] Solderitsch, J., Wickman, G., Kweder, D., Horton, M., "An Architecture and Generator for an Army IEW Domain," *Proceedings of the Seventh Annual Software Technology Conference*, Salt Lake City, Utah, 12 April 1995.